Programming Language Abstractions for the Global Network

Keith Sibson

Computing Science

Glasgow University

July 2001

A thesis submitted for the degree of Doctor of Philosophy.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract: Increasing demand for Internet-based applications motivates the development of programming models that ease their implementation. With the research presented in this thesis, we aim to improve understanding of what is involved when programming applications for the global network, and in particular the Web. We are primarily concerned with the development of language-level programming abstractions that address issues arising from the failure and performance properties of the Web. Frequent failure and unpredictable performance are ever-present aspects of any Web computation, so we must bring the properties of the Web into the semantic domain of our program systems. Our primary goal is to enable concise and intuitive expression of failure semantics in the context of concurrency, which is necessary for efficient Web computation given the large overhead in every network access.

The main scientific contribution of this thesis is the development of a Web programming model for which a major design goal is the integration of domain concepts, failure interpretation, concurrency, and a mechanism for flow of control after failure. Our model is the first to successfully achieve a clean integration. We develop a programming language called *Focus*, which incorporates two complimentary abstractions. *Persistent relative observables* allow reasoning about the dynamic behaviour of computations in the context of past behaviours. Examples of observables are the rate, elapsed time, and success probability of http fetches. The mechanics of our observables mechanism allows the generalisation of the observables concept to *all* computation, and not just Web fetches. This generalisation is key in our design approach to *supervisors*, which are abstractions over concurrency designed for the specification of failure semantics and concurrency for computations that contain Web fetches. In essence, supervisors monitor and control the behaviour of arbitrary concurrent computations, which are passed as parameters, while retaining a strict separation of computational logic and control logic.

In conjunction with observables, supervisors allow the writing of general control functions, parameterisable both by value and computation. Observables are abstract values that fluctuate dynamically, and all computations export the same set of observables. Observables allow genericity in supervisor control, since the mechanism constrains the value of observables within a pattern of fluctuation around a single number. Whatever the activity of a computation, information about its behaviour can be obtained within a range of values in the observables. This means that supervisors can be applied independently of knowledge of the program logic for supervised computations.

Supervisors and observables are useful in the context of the Web due to the multiplicity of possible failure modes, many of which require interpretation, and the need for complex flow of control in the presence of concurrency.

This work is dedicated to my mother Alison, who has supported me in more ways than she knows.

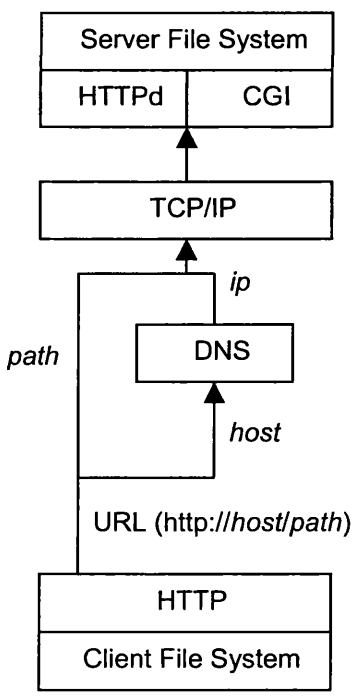
People to whom I am indebted for their help in aiding my research are my supervisor Richard Connor and Noel Winstanley. In particular, I would like to thank Richard, for his skillful guidance, and for teaching me how to be a scientist.

I composed this thesis by myself and the work it contains is my own. It was funded by an EPSRC studentship.

1: Introduction

Applications and the Web computational model

The Worldwide Web, WWW[1], or just the Web, is an enormous collection of files accessible on the Internet via the client-server query based *Hypertext Transfer Protocol*, http [1]. Many of the files available on the Web are documents written with the presentation-oriented *Hypertext Mark-up Language* (HTML), with which Web browser applications can appropriately display document content [2]. HTML documents can contain embed Universal Resource Locators (URLs), which are directed links to other documents or files anywhere on the Web [3]. It is the closure of all URL reachable resources available via http that constitutes the content of the Web, and gives rise to the visual metaphor of a Web of information. Recently, the Web has become synonymous with the Internet. In fact, the Web exists only over a subset of the physical Internet hardware, and at a layer of abstraction higher than that of the lower level Internet TCP/IP transport protocols. However, the Web is fast becoming ubiquitous, and today is certainly the most important facet of the global network.



The original computational model conceived for the Web centres around the integration of HTML *forms*, which when displayed in a browser allow input of textual information, and *CGI gateways* [4][5], which are interfaces to executable programs residing on Web servers. Browser applications encode all inputted form data as a single http query. This is then transmitted to the CGI gateway at the server, which decodes the query data and invokes the appropriate executable, which can be written in any programming language. The CGI gateway passes the entered form information to the newly invoked process, the output of which is streamed to the client browser as it is produced. In addition to textual input in fields, HTML also defines checklists and pulldown menus for inputting selection information. These give added flexibility in presenting application interfaces.

It is a simple arrangement, but the combination of CGI and HTML forms has proven to be a flexible model for implementing Web applications. The output of a CGI execution can itself

[1] WWW is the only known acronym that does not verbally abbreviate its associated phrase.

be a form, the structure of which may be dependent on the previously inputted form data. Although CGI is stateless, *hidden elements* in HTML forms allow the passing of data back and forth between server and client so that state can be maintained between form submissions. This means that transactions between server and client can span more than one interaction. In addition, the CGI executable may choose to store some of the inputted information long-term, locally on the server. However, the development of *cookies* [6], has allowed the storage of small amounts of state at the client-side, and has removed from the server some of the burden of storing large amounts of user-specific state in the long-term.

Microsoft's Active Server Pages (ASP) [7] and Sun's similar Java Server Pages (JSP) [8] are simplifications of the CGI and HTML forms computational model. Like CGI, all computation takes place at the server. However, the code for processing form parameters is more tightly coupled to the HTML document than with CGI, since it is physically embedded within the document. The server removes this code from the document and executes it before sending any HTML to the client. Thus, ASP and JSP are indistinguishable from CGI at the client end, except by observation of the filename extension in the URL. In addition to processing form data, ASP and JSP incorporate the functionality of 'server-side includes' [9] by default.

Web search engines such as Altavista [10] provide multiple form transactions in that once the results of a search are returned, the user can refine the existing search results according to new keywords. The server returns the original keywords to the client in hidden form fields, and these become part of the input for further form submissions.

Booksellers were among the first companies to realise the commercial potential of the Web. When buying a book through the Amazon [11] Web site, the user is presented with a multiple form interface in order to enter information such as credit card information, shipping details, and the like, and the inputted data is also entered into a database at the server.

During its execution the CGI application may itself access the Web. For example, Search Spaniel [12] collates and summarises the search results of over twenty different search engines, all from a single query.

To achieve prolonged interaction between client and server, queries and responses carry with them complete information about preceding state transitions. The repeated making and breaking of connections between client and server and the passing back and forth of information all incurs substantial overhead, both by communication latency, and by computational burden at the server. This overhead has motivated a paradigm shift towards a computational model where much of the input processing and verification takes place at the client side, thereby eliminating some CPU overhead at the server and latency. Although there is still a definite client-server relationship between the interface (client) and application

(server), the client becomes a logical extension of the server's CPU. Several programming languages have been developed that act as components of the Web browser application and are capable of client side computation. We describe three of the most prominent.

Programs written with Netscape's *JavaScript* language [13] are physically embedded in HTML documents, and provide a simple event-based computational model that may be used to program over elements of the html document and interface with the browser application. However, JavaScript has no concept of the world external to its host document, and so can be nothing more than a small client-side component of a larger Web application. Microsoft's JScript language is compatible with JavaScript except for a small number of nuances, and is supported by the Internet Explorer browser.

JavaScript should not be confused with Sun Microsystems *Java* language [14]. Java is a general purpose object-oriented programming language with which JavaScript has little in common. The main feature of Java relevant to the Web is that it has an *applet model*, where applications in the form of platform independent byte-code can be logically embedded within html documents. The applications reside on a Web server, and are transmitted to the client when the container document is downloaded. The application then executes within the context of the browser window. Security concerns prevent applets from accessing resources local to the client, and also prevent the application from directly communicating with any network host other than the server from which the application originated. In addition to applets, Java also provides a model for server-side computation based on the *servlet* [15]. Servlets aid the integration of applications with CGI, but unlike ASP they still rely on a loose coupling between the HTML form and processing code.

Microsoft's VBScript is a subset of the Visual Basic programming language. It has a similar computational model to JavaScript and JScript, but is reputedly easier to learn. The power of VBScript comes from the fact that it is primarily intended to integrate with ActiveX [16] components and ASP. ActiveX components (or controls) can be written with Visual Basic or C++, and perform the same function as Java applets. However, ActiveX components can be used in any context that supports COM [17]. Many Microsoft applications support COM, such as Word and Excel for example, so ActiveX Web components can be used in contexts other than the Web browser and visa versa. A drawback of JScript, VBScript, and ActiveX is that they are all proprietary technologies of Microsoft and as such work only in conjunction with Microsoft products. For example, as of summer 2000 the Netscape browser is used by 25% of Web users [18], but does not and is never likely to support JScript, VBScript, or ActiveX.

Java and JavaScript enhance the computational model of applications such as search engines, which reside on Web servers and present their interface globally. Such applications

are designed to interact with humans via a browser. However, there are many classes of purely client-side applications that treat the Web as a passive entity to be queried. The Web browser application is itself an example of this, albeit a somewhat superficial one since it concentrates on content presentation rather than computing over Web infrastructure. Perhaps the best known 'real' example of a client-side application is the Web *crawler* (or *spider*, or *robot*), which collects data from Web documents with little or no human interaction. Crawlers traverse the Web by recursively following embedded URL links. In general, all crawler applications involve concurrency, with several crawler processes cooperating in order to achieve greater throughput. Search engine crawlers extract and index the words of a document against its URL, but there are many other applications for crawlers, such as email address harnessing and link integrity checking, for example. There has been little work to enhance the computational model of these client-side applications, and traditionally they are programmed with general purpose (GP) programming languages such as C++ or Java (but not with applets or servlets), for example. Compaq's WebL is one of the few direct approaches to designing a language for this domain, and we discuss it in Chapter 3 – *Domain Properties and Flow Control.*

Mobile code can be considered to be any executable program that moves physical location before executing. With this loose definition, both JavaScript and Java qualify as having mobile code. However, JavaScript's computational domain is restricted, and the Java applet model is still server-centric in that the client CPU becomes a logical extension of the server. The World-Wide Web Consortium maintains a collection of links to information on mobile code systems [19]. Truly mobile code forms the basis of what is termed *agent based distributed computing* [20]. This is a vision of global computation partly due to Cardelli [21] where itinerant computational entities roam the Internet gathering information, performing tasks, and interacting with other agents. An example of an agent-based application is a Web crawler that physically moves its execution to a remote site in order to index content or perform some other task, thereby greatly reducing overhead incurred by connection latency.

The requirements of a distributed Web crawler best characterise the kind of issues we wish to address in the development of a Web programming model. A distributed Web crawler is a class of application that performs frequent Web access and has a requirement for robustness independent of human decision making. That is, the crawler agent must be able to automatically cope with problems such as intermittent server failure and broken links. Furthermore, since crawler mobility is motivated by performance and reliability issues, it must be capable of making mobility decisions based on observations of performance and reliability. A crawler agent should only incur the overhead of changing its locality if it anticipates improved performance or reliability. The requirements of this application domain

are similar to those of purely client-side applications, since a client-side application is an instance of a distributed agent that does not move. Since robustness and performance are equally important in many client-side applications, in developing a programming model for Web computation we take an approach that is general enough to encompass both domains.

Implementing agent based distributed computing over the Web requires abstraction over the existing protocols (http and TCP/IP) and interfaces (CGI). One way to achieve this is by the development of new higher-level protocols and interfaces. However, this approach is problematical because the Web is an autonomous distributed system with a firmly entrenched software infrastructure of server installations. To adopt new protocols and interfaces that provide a distributed model of computation requires the installation of new server software for all hosts taking part. Moreover, it is unlikely that a single protocol or interface could cater to the demands of all, since the most appropriate computational model may differ widely between applications. A more flexible approach is to build applications that rely on the standard protocols and interfaces of the Web, but abstract over them at the language- or application-level. This allows applications to adopt an appropriate level of abstraction for their purposes, at the API- or language-level. For example, an agent based distributed system might consist of several servers each exporting a CGI application that sends and receives executable agents. The protocols that govern security and the marshalling and unmarshalling of agent entities are not dictated by the server, but by abstractions within the application program itself or in the language with which it is written.

JavaScript scripts and Java applets have simple mobility models that are not general or flexible enough to express global computation by themselves. Thus, they are useful only as part of a model for global computation. General purpose programming languages have the capability, but are not geared specifically to the task. As a result, there has been much research into the development of programming models specifically for agent-based global computation [22][23]. However, these models, along with general purpose language models, JavaScript scripts, and Java applets all fail to address the issues raised by the inherent failure and performance properties of the global network. This makes it difficult to program robust applications. It is clear that agent based computing will be founded on programming models designed specifically for the computational medium. Due to the ubiquity of the Web, it is perhaps the most appropriate infrastructure upon which to build models of global computation. This thesis is an attempt to aid the realisation of global computation by investigating programming models that directly address the issues raised by failure and performance on the global network.

Failure and Performance Properties of the Web

This thesis is concerned only with failure and performance issues that arise when automating tasks in the Web domain, and not with automation over Web content. Content formats change. However, the inherent failure and performance properties of the Web do not, since the Internet will always be subject to largely unpredictable fluctuations in bandwidth, node failure, and unenforceable referential integrity.

Before we continue, it is worth pointing out that there is an important class of failure that does relate to the structure of Web documents. Since Web servers are autonomous (decentralised control), document hierarchy and individual document structure is beyond the control of clients, and is subject to change without notice. If the structure of a Web document changes, it is barely an inconvenience for human browsers. If a document moves to a different location, human browsers can usually locate it again without too much inconvenience. In contrast, uncontrolled format changes or document relocations are potentially catastrophic for automated agents that are dependent on those documents. This is because development techniques traditionally applied to applications dependent on persistent data cannot be appropriately applied in a Web context. For example, programmers writing applications that use files stored on local hard disk usually assume that if a file was previously written with a specific schema, then it can be read according to that same schema. However, Web host autonomy means that the structure of Web documents can never be presupposed, since there is no universal schema for Web data. The issue of enforcing or deriving structure for Web documents to allow automated processing is a large subject, and is beyond the scope of this thesis. We refer the interested reader to introductory material on semi-structured data [24][25], XML [26], and the Document Object Model [27].

The Web exists as a layer over the Internet, and does not mask any of the failure or performance properties of the underlying network substrate. In fact, it introduces several new failure modes at the http level, such as document not found, for example. In general, for every Web query there is non-determinism as to,

- whether the query is valid (existence of host, server, and resource),
- how long the query will take to complete, if at all,
- the transfer rate and how it will vary over time,
- how long it will take the server to respond initially (latency), and
- whether any failures experienced are temporary or permanent.

To better understand the failure and performance properties of the Web, we examine its major working parts. There are five main components that constitute the Web[1]:

- *URL – Uniform Resource Locator.* URLs identify single resources on the Web, but not uniquely (more than one URL can identify the same resource). URLs consist of protocol, hostname, and resource path parts.
- *HTML – Hypertext Mark-up Language.* This is the presentation-oriented language with which much of the Web's content is written. HTML documents can contain embedded URLs that link to other resources on the Web.
- *DNS – Domain Name Service.* This is a mechanism to convert textual hostnames into IP numbers, which form the underlying address space for the Internet [28].
- *HTTP – Hypertext Transfer Protocol.* Http is a query-based protocol that specifies how Web servers and clients communicate with each other once a connection has been made.
- *TCP/IP – Transmission Control Protocol/Internet Protocol.* Http queries and responses take place via socket connections between the client and Web server. These socket connections are made and broken, and data transmitted across them, according to TCP/IP.

We are concerned only with the failure and performance properties of the Web, and the components that directly relate to these are http and TCP/IP. Http defines the structure and formatting of Web query messages that are sent to servers and of the responses that are received by the client. Underlying this is TCP/IP, which is concerned with making and breaking connections, and in sending and receiving bytes across those connections. Http is an abstraction over TCP/IP, since it specifies a default port (80) for socket communication, and restricts data flow to a single query followed by a single response. However, the nature of the underlying socket is still exposed. Thus we can easily measure transfer time and connection (latency) time, and calculate transfer rate as data is streamed across the socket.

It is these measurable aspects of TCP/IP sockets as they relate to http that allows us to reason about the failure and performance properties of individual Web queries. We have performed an examination of the properties of many queries, across several geographically

[1] Most of these are shown in the diagram at the beginning of this chapter.

diverse Web servers and at different times of the day, in order that we may draw some conclusions about the performance and failure properties of the Web as a whole. We also draw from experimental data provided by Zeus Technologies [29]. The details of both experiments are somewhat complex, so we describe the methodology and analysis only briefly here, going into more detail in chapter 2 – *Analysing Web Failure and Performance.* The conclusions we have drawn from these experiments are important in justifying our approach to providing abstraction for Web programming.

Zeus Technologies are undergoing a continuous study of the availability of 241 web sites. Availability is defined as being able to connect to the server and download the front page within sixty seconds, not including images. Sites under test are checked for availability every fifteen minutes, which equates to nearly 3,000 tests a month. So far, the experiment has been running for two years. Availability on a month per month basis ranges from 0% to 100%, but is on average around 98%. From the Zeus data, we conclude the following.

- Failures are frequent and intermittent.
- Failure by dropped connection is extremely rare.
- Connection failure (timeout) is the most common type of failure[1].

Our own experiments concentrate more on the performance aspects of Web fetches. With an average sampling granularity of fifteen minutes, we gathered data over 24 hour periods for latency, download time, average download rate, and fluctuations of rate over the download duration. We chose twelve servers for their diversity in geographical location, server software, expected usage patterns (load), and perceived bandwidth. Our conclusions can be summarised as follows.

- Median latency is a function of geographical distance.
- Anomalies in latency motivate at least one retrial for *every* timeout.
- Perceived bandwidth is inversely proportional to distance to a small degree.
- Bandwidth fluctuates throughout transfer.

[1] Excluding Http 404, Document not found. In classes of application such as Web crawlers, this is likely to be the most common form of failure.

- Bandwidth fluctuates more during periods of network congestion.
- Bandwidth fluctuation is caused by server and network load.
- Average rate for distinct transfers is consistent in the short term, but shows definite trends according to the time of day.
- Transfers from the same server will achieve similar average rates given similar network and server load.
- Rate 'troughs', where transfer rate drops to zero, are common, even when the network or server is not under particularly heavy load.
- Troughs are more frequent, and of longer duration when the network or server is heavily loaded.
- Troughs can range in duration from several seconds to several minutes.

Since the global network is a collection of distributed, autonomous nodes, it is inherently unreliable and exhibits unpredictable performance. Practical analysis confirms this. However, there is a theoretical result due to Cristian that is perhaps more disconcerting [30]. Given any system with distributed autonomous nodes, in principle it is impossible to distinguish between failure and a network link or server that is very slow. A corollary to this is that the failure of a network link cannot be distinguished from failure of the server at the end of the link. These properties have major implications for all distributed applications that attempt failure detection. Traditionally, failure detection is achieved by timeout. However, no value for timeout can ever be entirely reliable, since there is always the possibility that tardiness on the part of the server or network link will cause an incorrect interpretation of failure.

Distributed applications must be able to operate in the face of this non-determinism with minimal guiding human interaction. One approach to this is to provide a level of distribution abstraction by having the programming model mask the non-determinism. Why this is inappropriate is the subject of the next section.

Distribution Abstraction

In designing distribution abstractions for the global network, there are three possible approaches. The first is to provide transparent distribution abstraction that masks the failure properties and non-determinism of the network. Essentially, this approach attempts to merge the computational models of distributed and local computing by making all computation follow the model of local computation, and ignoring the different failure modes and

indeterminacy inherent in distributed computing. Both Obliq [31] and the Linda co-ordination model [32] attempt this. However, by ignoring failure and indeterminacy, systems produced with such programming models are unreliable and are incapable of scaling beyond small groups of co-located machines that are centrally administered. Local computational models cannot be applied to the global network, since distribution issues such as latency, failure, and autonomy are intrinsic aspects of the domain. If we cannot mask these properties, then we must expose them.

The second approach is to provide transparent distribution abstraction by uniformly exposing the failure properties and non-determinism of the network in *all* computation. That is, all computation resembles distributed computation, even if it is local computation that is taking place beneath the abstraction. CORBA [33] and Emerald [34] are distributed programming models that attempt this, in that the object interfaces are defined independently of object locality. Although this approach scales well, it requires programming practice that is far removed from that of local programming, and makes local computing more complex than would be otherwise necessary. For example, in subsuming the local programming model with that of distributed programming, every single reference must have the characteristics of a distributed reference. This means that the interface to all objects and values in the system must be designed so that the objects and values react in a consistent way to partial failure. Likewise, the interfaces to all objects must inherently be designed for concurrency. These are unnecessary restrictions on objects that are local.

Both of these approaches attempt a transparent integration of distributed and local computing. However, this is difficult because the computational models of distributed and local computing have irreconcilable differences. Primarily these are a result of non-determinism in distributed computing as compared with the determinism of a closely coupled architecture. Although it is logically possible to paper over the difference between local and remote access, problems introduced by partial failure and concurrency seem to indicate that such unification is impractical. Waldo provides compelling arguments that support this, suggesting that the only viable way to provide distribution abstraction is to relax the requirement for transparency, so that the programmer becomes aware of the differences between local and distributed computing. He states in [35] that:

> "The reality of partial failure has a profound effect on how one designs interfaces and on the semantics of the operations in an interface. Partial failure requires that programs deal with indeterminacy. When a local component fails, it is possible to know the state of the system that caused the

> failure and the state of the system after failure. No such determination can be made in the case of a distributed system. Instead, the interfaces that are used for the communication must be designed in such a way that it is possible for the objects to react in a consistent way to possible partial failures."

Thus, the distributed nature of objects must be reflected in their interfaces. Java RMI [36], SR [37], and Occam [38] are all programming systems in which distribution is explicit, and provide some form of abstraction, such as the automatic marshalling of parameters to remote procedure calls, for example. The many different approaches to distribution abstraction each have their advantages and disadvantages, and no specific approach is generally accepted as the most appropriate. This suggests that a particular distribution abstraction cannot be all things to all people. Moreover, all of these systems require the installation of special server software that implements the relevant protocols and interfaces. We have already noted that a more flexible approach is to use already widely adopted protocols and interfaces, namely those of the Web.

Our main concern is that existing programming models for distributed computing do not address the issues of failure and performance for the Internet identified in the previous section, and as a result cannot scale globally. At best, the systems allow the specification of timeout in order to interpret failure, and embed an exception handling mechanism that can be used to implement flow control after failure. We see the issue of providing distribution abstraction as being distinct from that of dealing with the properties of the domain. Indeed, we argue that they are in tension with one another, because instead of hiding the failure and performance properties of the Web with abstraction, they should be exposed, allowing the explicit programming of failure models.

It is problematical to model high-level concepts such as distribution directly, due to the difficulty in masking the failure properties of the global network. We do not wish to develop yet another distribution paradigm that enforces a particular computational model. Instead, we want to provide a level of abstraction that exposes the properties of the domain and allows the *implementation* of arbitrary higher-level distribution abstractions if necessary. Consequently, we take a lower level approach, performing a more direct attack on the intrinsic failure properties and non-determinism of the Web.

Human Browsing

Humans have evolved behavioural means to ease the performing of manual tasks on the Web. In this context, a task might be to find information about a specific subject, or buy a

particular CD for the cheapest price possible, for example. By examining the methods that humans employ to achieve them efficiently, we hope to gain some insight as to how such tasks might be automated. In particular we are interested in how humans deal with the failure and performance properties of the Web. There has been some research into human browsing and task models for the Web [39][40]. These are HCI studies that are primarily concerned with improving the design of Web browser applications. The tasks that test subjects perform tend to be simple, and the studies concentrate on interaction with the browser application more than interaction with the Web as a concept. Neither study addresses the issues of failure and performance, which are our primary concerns. We have been unable to locate any studies that establish human thought processes with respect to Web failure.

The design of a high-level programming language is chiefly about easing the mapping of human thought processes into something that can be understood by a computer. Thus, we feel that the insights gained from examining human 'algorithms' for Web 'computation' are extremely valuable in ascertaining what concepts should be provided in Web programming language abstractions. We are interested in human thought processes with respect to Web failure and performance, and in particular how failure is interpreted and in how task flow is affected by failure. We have undertaken a study of these issues, but it is somewhat philosophical, in that it is entirely based on introspection, and on informal discussion with experienced Web users. We lacked the resources for extensive HCI experiments. However, since any insights gained from our study are to be used only in guiding the design of programming abstractions, and not as a foundation for design, we feel that intuition alone is sufficient. In any case, we aim to provide as general a mechanism for failure interpretation as possible.

Human browsers frequently employ concurrency in order to achieve greater throughput. However, the level of concurrency is dependent on local bandwidth. Users with permanent connections with high local bandwidth tend to download more things concurrently, such as large archive files, for example. This increases overall throughput, since in general the bandwidth bottleneck does not lie local to their machine. In addition, since their connection is permanent, they are not concerned with utilising the bandwidth at all times. They are interested only with saving their own time, and not in minimising 'online' time. Modem users use concurrency in a different way. Downloading several things concurrently via a modem connection decreases the overall throughput, due to multiplexing over the line. The bandwidth bottleneck is local. To avoid this, modem users try to download things sequentially. Also, as a reaction to the fact that they are not paying for the number of bytes transferred, but for the amount of time spent online, they try to keep the bandwidth utilised at all times in order to minimise the total online time required in completing the task. For example, when

downloading a large archive split into six parts, say, the high bandwidth user will download all parts at the same time, and unpack them after all have downloaded. In contrast, the modem user will download them sequentially or perhaps two at a time, and unpack them as they arrive.

Since modems are slow, humans tend to perform other tasks while downloads proceed. For example, when reading a document that is split into sequential parts, modem users will read the current page while downloading the next, whereas high bandwidth users will download the next page only when they finish reading the current one. In general, users with high bandwidth are more likely to wait idly while downloads proceed, since there is not enough time to merit a 'context switch' to initiate another action. Our conclusion is that human browsers employ sophisticated control over the level of concurrency, in computation and data transfer, in order that it is the most appropriate for their purposes and for their bandwidth.

Concurrency is also employed as part of the human browsing 'failure model'. The most common model for failure in general is the sequential 'on failure do x instead'. For example, when accessing their favourite Web site for a particular kind of information, news, say, the user has a list of alternative sites containing the same or similar material. On failure of the preferred site, one of these is invoked. However, human browsers often employ concurrency here. If their favourite site looks like it *might* fail, then a secondary site is invoked. Whichever downloads first is the one read. Pessimistic users can be seen to invoke secondary downloads speculatively, at the same time as the primary, taking whichever completes first. In general, we define a class of behaviour called *control behaviour*, which encompasses the following kinds of behaviour.

- 'On failure do something else' where the else may be an equivalent process to one that failed, or something unrelated.
- 'Alternate actions' where two or more downloads or computations are started simultaneously and whichever completes first is taken as the result, the other being terminated.
- 'Independent concurrency' where two or more unrelated downloads or computations are performed at the same time.
- 'Related concurrency' where one or more downloads or computations are invoked as a result of observations of the behaviour of one or more ongoing downloads or computations.

In essence, the human 'skill' of browsing is the diagnosis of failure from visible 'symptoms' associated with Web downloads in the context of several other factors, then the taking of appropriate action. Those symptoms we can quantify we term as *observables*. These are:

- Transfer rate.
- Download time.
- Connection latency.
- Amount of document downloaded so far.

These observables are used in isolation or together in order to interpret failure. Human browsers have a vague notion of 'timeout' for Web transfers, and may take remedial action if a fetch does not complete within an acceptable time. However, a human browser is much less likely to terminate a transfer if it is close to completion, and is likely to take into account transfer rate. Human browsers use arbitrary combinations of observables when interpreting failure. Other factors that are taken into consideration when interpreting failure are as follows.

- Time of day.
- Server locale (geographical distance).
- Perceived server load and network congestion.
- Level of importance attached to the information being downloaded.
- Previous reliability of server or URL.

These are factors that are not directly observable from the properties of an ongoing fetch, but can be important in interpreting failure in conjunction with observables. Human browsing behaviours are a reaction to the failure properties outlined in the previous to last section, but there is something more, something holistic going on here that we want to capture. The rate of a Web transfer may quicken, slow down, or even drop to zero. Experienced human Web browsers are good at interpreting these symptoms, and may terminate the transfer, retry it, or seek alternative sources of the same information. It is knowledge of their previous browsing experiences with that particular site or URL that aids in interpreting the symptoms of failure and in determining an appropriate response. Human browsers interpret failure based on

observations of the properties of particular fetches, in a context of past behaviours, time of day, and other factors.

What is important is that the human browsing model is fundamentally based on the *perception* of failure. This is the type of failure we are interested in, since we see it as an area of weakness in contemporary programming models, which have difficulty in capturing 'vague' notions of failure based on context sensitive observation. Most programming languages are designed to operate within the more deterministic context of a closely coupled architecture, where failure is generally absolute. New programming models must be sought that capture this holistic notion of Web failure and performance.

Programming the Web

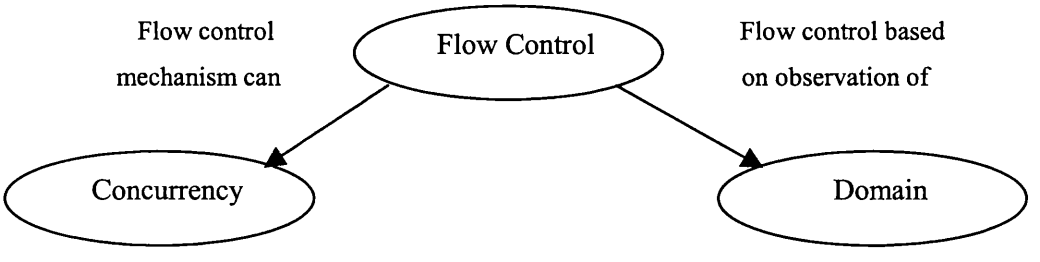
Contemporary general purpose programming languages are designed to write applications that execute over local file systems. Failures arising from file-system access are absolute, and are usually repeatable. In contrast, failure in accessing the Web is intermittent, and is not absolute in that it often requires interpretation, given a number of factors. When executing over a file-system, the existence of a program's dependent files is usually a precondition for successful execution. In contrast, the frequency and intermittence of failure on the Web means that enforcing program preconditions with respect to Web access is not viable. For example, on host lookup via a Domain-Name Server (DNS), an intermittent error may result in failure to resolve a particular hostname, even if the host does exist. In contrast, with a local file-system a failed attempt to open a file almost certainly means that the file does not exist. When reading a file from local disk into memory, programmers rarely consider the time that the read operation will take to complete, since although it is non-deterministic, it is usually so small as to be irrelevant. The time taken to download Web documents is also non-deterministic, but is certainly orders of magnitude longer than reading documents of similar size from the local file system.

Although determinism is technically not a 'fuzzy' concept (a computation is either deterministic or non-deterministic) we use it in a fuzzy context here. If something is more deterministic, it is more predictable. Computing over the Web is less deterministic than over a local file system, since the relationship between a Web application and the Web medium is fundamentally different to the relationship between a traditional application and the underlying file-system. In spite of this, designers of Web programming abstractions seem determined to view the Web as a file system. For example, the java.net package [41] provides a Web fetch abstraction that produces a data stream indistinguishable from one produced by file system abstractions. No facility for examining rate or timing latency is provided by the abstraction, and must be explicitly programmed if required. In general, the fact that

contemporary programming models are geared towards computing over a local file-system means that the abstractions they provide tend to be inappropriate for computing over the Web. That is, the underlying semantic space for these general purpose programming languages does not reflect the properties of the Web. Forcing programmers to deal with a broad class of non-deterministic behaviour introduces complexity because these properties of the domain must be mapped into a language whose semantic space is very different. This exacerbates programmer errors.

In the earlier section on distribution abstraction we argued that it is inappropriate to provide high-level distribution abstraction in a language designed for general Web programming. We aim to provide a level of abstraction that is as flexible as possible, so do not attempt to model any distribution other than the most primitive actions – Web get and post. Any higher-level notion of distribution would be likely to mask the properties of the domain in some way, properties that may be useful to some classes of application. Since Web get and post are primitive actions, they should be provided as primitive operations in a Web programming language. We have argued that the nature of the domain must be exposed, so if the language incorporates primitive Web fetch operations, then the domain exposure must be associated in some way with these operations. There are two other concepts that we require of a Web programming model.

Firstly, the significance of network overhead motivates concurrency, since we wish to perform useful work during downloads. This motivation is additional to that of being able to express the desirable Web computation behaviours exhibited by human browsers. Human



browsing behaviour shows us that we require concurrency for Web fetches and arbitrary computation. For example, human browsers might read one document while another is downloading, and this has a direct analogue in automated processing. Secondly, intermittent and frequent failure (compared with a file system) motivates the adoption of appropriate flow control mechanisms for failure. The flow control mechanism must allow behaviours similar to those of human browsers. For example, a human browser may initiate a concurrent download after observing an unacceptable slowdown in another. To this end, the flow control mechanism must integrate with the concurrency mechanism.

To summarise, a Web programming language requires primitive Web fetch operations, exposure of the domain properties somehow associated with these operations, concurrency for Web fetches and arbitrary computation, and a flow control mechanism that integrates with the concurrency mechanism basing flow control decisions on logic involving quantified domain properties. The relationship between these concepts is shown in the diagram above.

This Thesis

As the Web becomes more important both economically and culturally, there is a growing demand for sophisticated application programs in the domain. These applications must be able to compute over the Web in the face of its non-deterministic failure and performance properties, with little or no human interaction. This gives rise to a problem domain that is the development of a class of Web applications for which robustness is key. To program in this domain, we need a semantic space that reflects the failure and performance properties of the Web, constructs for flow control that are appropriate given the frequency of failure, and a mechanism that provides concurrency for downloads and arbitrary computation. Of utmost importance is that these three concepts integrate appropriately. Our approach involves the design of a domain specific Web programming language, *Focus*, which is the first language to successfully integrate these concepts. This is the key scientific contribution of our thesis, though we allude to the fact that Focus provides the means to express Web computation more concisely and more intuitively than contemporary programming languages, as well as being more flexible in expression.

Our approach is split into two parts. First we define a conceptual domain that exposes the failure and performance properties of the Web. This conceptual domain is based on *persistent relative observables*. These are quantities associated with ongoing computations (Web fetches) that reflect the performance of those computations with respect to previous behaviour. For example, we define a rate observable that is calculated as a ratio to a historical average rate for that particular Web server. Second, we define a high-level language construct called a supervisor, which we embed in a programming language that incorporates our conceptual domain. Supervisors are abstractions over concurrency that allow the expression of computations that control other computations based on queries of their persistent relative observables. In essence, supervisors effect concurrent flow control based on a programmed interpretation of the significance of observables for computations passed to them as parameters.

In Chapter 2, *Analysing Web Failure and Performance*, we present the methodology and results of an experiment to determine the failure and performance characteristics of the Web. The experimental data confirms what is generally already known: the Web is failure prone,

and subject to fluctuations in performance. However, it also exposes some interesting performance patterns that influence the design of our programming model for the Web.

In Chapter 3, *Domain Properties and Flow Control*, we outline some methods of flow control for failure, including function return code overloading and exception handling. Then we describe in detail the language WebL, and the Service Combinator Algebra, both of which attempt to provide abstraction over the failure and performance properties of the Web by integrating means for failure interpretation with exception handling and concurrency. At the end of the chapter, we identify three properties we deem as desirable for a certain class of applications involving the Web, and evaluate Service Combinators and WebL in context.

In Chapter 4, *Web Fetching with General Purpose Languages*, we describe a methodology that provides flexible failure interpretation for Web fetches without recourse to specialised programming language concepts. The methodology makes use of higher order functions to parameterise Web fetches with expressions that dictate the conditions for failure. In the interest of pragmatism, we also present an approximation to the technique for object-oriented languages. We claim that our methodology is a useful tool for programming Web applications with general purpose programming languages, in particular with respect to failure interpretation, but admit that it has difficulties in integrating with concurrency mechanisms. Primarily, this is a result of the serialised nature of exception handling mechanisms.

Chapter 5, *A Conceptual Domain for Web Programming*, describes a conceptual domain for Web programming based on the concept of persistent relative observables. A conceptual domain is the set of primitive concepts upon which programming languages are defined. In our domain 'observables' such as transfer rate and time are calculated relative to a historical context. The benefits of persistent relative observables include program portability, mobility, and future proofing, as well as providing the means to define programming languages that have flexible, efficient, and accurate failure interpretation.

In Chapter 6, *Observation and Control with Supervisors*, we introduce supervisors, which are high-level language abstractions that are appropriate for computing over the persistent relative observables conceptual domain. In essence, supervisors are concurrency constructors, and rely on the observables of the concurrent computations they control to interpret failure or other conditions that require special processing. Major features of the supervisor mechanism are the separation of computational logic from control logic, independence of control from the type of computations being controlled, and a novel mechanism for resolving concurrent updates.

Chapter 7, *Exception Handling*, is an in depth study of exception handling, which is the flow control mechanism for failure universally adopted by modern general purpose programming

languages. We identify the aspects that may differ between mechanisms as being exception form, flow of control after detection, level of automatic propagation, type of exception interface, and method of exception binding.

In chapter 8, *Related Work*, we describe work that is not directly related to programming the Web but is still relevant in that it involves issues of abstraction with respect to failure or performance. We also describe work related to programming the Web but which does not address the issues of failure and performance.

In chapter 9, *Formal Issues*, we prove by simulation that the supervisor construct conceptually contains the service combinator algebra. We also present an algorithm that implements the supervisor environment model efficiently and prove its correctness.

Chapter 10 concludes this thesis by summarising its content and scientific contribution, as well as outlining some areas of possible future research.

2: Analysing Web Failure and Performance

In this chapter we present the methodology and results of two experiments that determine the failure and performance characteristics of the Web. First, we examine an undergoing study by Zeus Technologies [42] that tests the availability and performance of nearly 300[1] Internet Service Provider (ISP) Web sites across the world [43]. Then we introduce our own experiment that concentrates more on the performance aspects that are not fully addressed by the Zeus study. Finally, we present detailed conclusions.

The *failure properties* of Web queries are characterised by the frequency and patterns of failure occurrence. The Web has many failure modes. These are the union of the failure modes of the http protocol and of TCP/IP socket streams [44]. All http failure modes are absolute, and can be categorised into failure due to an incorrect query on the part of the client (malformed URL, requested document does not exist, etc) and failures on the part of the server (too busy to fulfil request, misconfiguration or internal error, etc). There are several absolute TCP/IP socket failure modes. Most absolute TCP/IP errors are local in nature, such as out of memory and insufficient local stream resources available, for example. Such errors are rare. The socket protocols optionally keep connections 'warm' in the absence of any other activity by forcing null transmissions roughly every minute. However, sockets will automatically return an absolute error if at any point the socket does not respond within a particular time. This time is not specified, but the protocol documentation suggests approximately five minutes, which is deemed sufficient to unambiguously indicate socket failure. It is the implementers of the operating system that are responsible for deciding the length of this 'absolute' timeout.

The *performance properties* of Web queries are those properties of Web fetches that are observable and quantifiable, and change according to factors such as network and server load. We have identified transfer rate, connection latency, and transfer time, all of which derive from the TCP/IP layer. However, we choose to exclude transfer time from our measurements, since it is dependent upon the size of resource being downloaded, whereas latency and rate are not[2]. The performance properties of the Web are important, because many classes of TCP/IP 'failure' manifest themselves in the form of no response, and cannot be distinguished from each other. Taking a naïve approach to this, we can rely on the socket timeouts provided by the implementers of operating systems. However, for the class of applications we are interested in, such timeouts are inflexible and inefficient. To capture 'undetectable' failure

[1] The Zeus study is expanding over time, and the number of sites involved is increasing.

[2] There is, in fact, a potential relationship between rate and resource size, which we describe later.

modes such as failure of the server to respond due to software, machine, or network failure we must interpret failure by using means more sophisticated than timeout alone. This motivates an analysis of the Web's performance properties, in order to determine how they can help in interpreting failure.

Throughout this thesis, interpreted failure is our primary concern. Absolute failure modes introduce similar problems to those encountered frequently in traditional programming domains. Thus, the means to deal with this kind of failure are well understood. For example, exception handling is an abstraction for dealing with absolute failure, and we survey the available mechanisms in chapter 8. In contrast, the concept of interpreted failure mode is alien to traditional programming practices, and the purpose of this chapter is to confirm that it is this kind of failure mode that is of most use when programming in the Web domain.

Zeus Study

According to the Zeus Technologies Web performance site, the purpose of their study is,

> "To measure and record the level of service providing by web hosting companies. Our only concern here is to measure the availability and performance of the http services offered by these companies."

All hosts are the Web servers of Internet Service Providers (ISP) hosted in the United Kingdom, and there are separate studies of these servers from hosts in the UK and in the USA. We have independently verified that the choice of target servers reflects a diversity of geography (albeit within the UK) and server types. We assume that since the study shows diversity in transfer rate performance, this reflects diversity in bandwidth provision.

> "This should provide a useful resource for companies looking to outsource their web hosting as well as a useful benchmark for the web hosting companies themselves."

Zeus technologies produce http server and load balancing software products, and the performance of such software can relate directly to the measurements in the Zeus study. However, the study does not attempt to correlate performance data against the type of server software in any way, suggesting impartiality. In any case, the Zeus study provides only data, and any analysis presented in this section is our own.

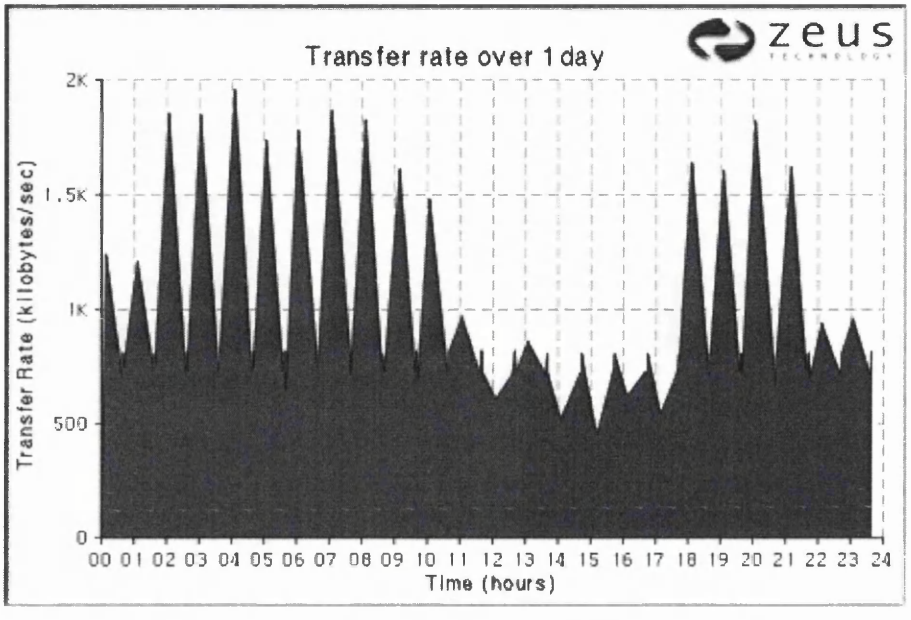
The Zeus study consists of two parts: failure properties experiments and performance experiments. First, we concern ourselves with failure. This study examines approximately 300 servers in the UK, from two hosts in the UK. The two hosts query alternately, approximately every fifteen minutes. This equates to nearly 3,000 tests a month for each target server. So far, the study has been accumulating data for over two years, though the number of sites under test is growing, so not all sites have two years of data. 'Availability' of a site is defined as the download of its front page (root URL) not failing. The survey classifies three different possible failure modes:

- *Connect* – No connection could be formed to the server. This failure mode encompasses manifest failure of the intermediate network link, failure to find an IP address via DNS lookup, and the server refusing the connection.
- *Http* – A connection was formed with the server, but it sent an invalid http response, an http status code not between 200 and 399 thus indicating an error, or the server broke the TCP connection before all data was received.
- *Timeout* – A connection may or may not have been formed, but the client was unable to resolve the DNS name, connect to the website and download the front page (excluding images) within sixty seconds.

In general, Web server availability on a month per month basis from UK testing hosts ranges from 0% to 100%, but is on average around 98%. There are no availability figures from the USA. 76% of all failures that occurred were those of timeout, and the vast majority of the remainder were absolute connection errors. Very few http errors were encountered. This suggests that it is 'undetectable' errors in the form of indefinite delay that are the most significant type of failure on the Web.

The second part of the Zeus study examines the transfer rate of the same 300 servers over time. Average transfer rate across the entire duration of a fetch is calculated for the download of a resource from each target. Image resources are chosen since they tend not to be the result of CGI processing, which would give rise to invalid rate measurements. Furthermore, images of similar size are chosen in order to eliminate any differences in transfer rate that arise from downloading resources of varying size. As with the availability tests, two different client hosts in the UK take part, but there is an additional host in the USA. The UK hosts alternately analyse the performance of target sites approximately every fifteen minutes. The USA host tests at a larger and more variable granularity.

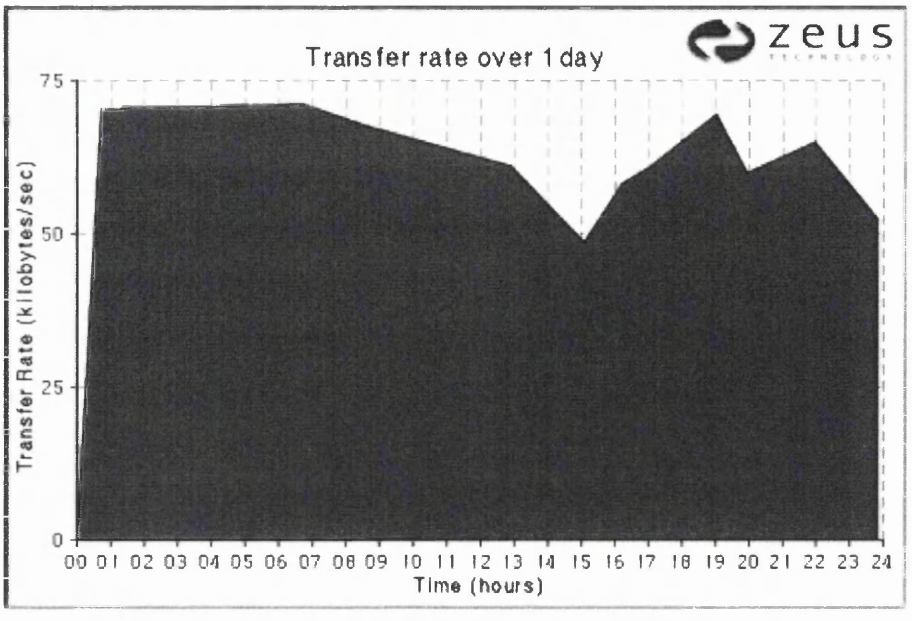
When testing from the UK, the fact that there are two different client hosts testing alternately gives rise to rate graphs that are spiked. This reflects the fact that the two testing hosts have different local bandwidths, and in some cases this is the bottleneck.



[www.inweb.co.uk 30th June 2000 from UK]

An observation from this is that during daytime hours the bottleneck moves from local bandwidth to intermediate network bandwidth. The spikes for the high bandwidth host fall, but the transfer rate for the lower bandwidth host remains broadly the same. Thus, proportionally speaking it is higher bandwidth local connections that suffer more due to fluctuations in bandwidth arising from the time of day. This suggests that if an application on a particular host is expected to perform most of its Web access during the day, it is probably not worthwhile investing in a very high bandwidth connection.

Testing from the single host in the USA shows no spiked effect other than that which might be attributable to time of day, as shown in the graph below.



[www.inweb.co.uk 15th May 2000 from USA]

From the graph, we can see that the granularity of testing is much larger for the testing host in the USA. The Web site does not state why this is so, though we suspect that it is because a single host must test the same number of sites, and also takes longer to process each individual site due to the lower average transfer rate across the Atlantic.

In the USA graph, there is some rate fluctuation due to time of day, though it is not as marked as with high bandwidth clients in the UK. The Inweb server is consistently very fast when accessed from the UK. In addition, the host in the USA has a high local bandwidth. Thus, we can conclude that the lower bandwidth exhibited by transfers from the UK to USA than by transfers from the UK to UK is an artefact of either geographical or topological distance in the network, or both.

One limitation of the Zeus study is that all of the target hosts reside in the UK. In contrast, the classes of application we are interested in are expected to exhibit Web access patterns that are much broader in terms of geographical diversity. Failure and performance measurements for sites in the UK are unlikely to give a true reflection of the properties of the Internet as a whole. Although the Zeus study incorporates some testing from a host in the USA, this is not sufficient diversity for our purposes.

Another issue that arises from the class of applications that we are interested in is interpretation of failure from available information. This is fundamental. Thus, we are interested in all available Web performance properties as they relate to failure. In this respect, one shortcoming of the Zeus performance measurements is that they do not include connection latency.

In 1995, Bob Metcalfe predicted the imminent collapse of the Internet due to increasing numbers of users outstripping the capabilities of the network infrastructure [45]. From 1994 to 1996 the number of Internet hosts almost quadrupled, from 2.2 to 9.5 million [46]. However, ongoing studies by Matrix.Net indicate that there was a 30% *reduction* in aggregate Internet latency over this period [47]. The trend towards lower latencies continues, indicating that Metcalfe's prediction is unlikely to ever occur.

Latency will always be an important aspect of any network access, since it is the only performance property for which there is a physical bound, namely that dictated by the speed of light[1]. In the future, data rates may be so great as to make transfer time insignificant. However, connection latency (or response time), will always be bounded by the amount of time it takes a request to travel at the speed of light to its destination and for the response to return. Knowledge of the global network's latency performance characteristics will be critically important if the expected trend towards mobile computation occurs. In the high bandwidth networks of the future, latency is likely to be the primary factor degrading the performance of distributed applications. Mobile components of distributed computation can ameliorate performance by migrating in order to minimise communication overhead.

The human browsing model indicates that observations of dynamic transfer rate during a Web fetch is useful in interpreting failure. The transfer rate of Web fetches can and often do fall to zero after a time, and may or may not resume. The Zeus study only calculates the average transfer rate across the *entirety* of a Web fetch, by dividing the size of the resource by the total time taken to download it. In contrast, we are interested in rate at a more fine grained level, in that we want to understand how rate fluctuates across the duration of individual transfers. The lack of latency, dynamic rate analysis, and limited geographical diversity in the Zeus study motivates us to perform our own experiments.

Our Performance Study

Our approach concentrates mainly on the performance aspects of Web fetches. However, we do record failure when it occurs, and we adopt the same three failure modes of the Zeus experiment except that our hard timeout is longer, since we use larger target resources. Although the Zeus study of failure patterns is extensive, it is not comprehensive since failure patterns for sites in the UK might differ from those of servers more geographically distant. We discuss failure patterns as they relate to geography later in this section.

[1] Network infrastructure is tending towards fiber-optic, so we use 'light' rather than 'electromagnetic impulse'.

Each of our experiments involves the repeated download of resources on twelve different servers over a 24-hour period. We performed two sets of experiments, from client hosts with different local bandwidths, but both of which are in the UK. One client is located on the Joint Academic Network (Janet) [48] and has a local bandwidth bottleneck of 10Mbit. Janet has extremely high bandwidth backbones within the UK. The other client has cable modem connectivity, with a local bandwidth bottleneck of 512Kbit downstream and 128Kbit upstream. Our cable service provider, NTL [49], is corporate and so does not use Janet infrastructure.

The target servers used in our experiments are chosen for diversity in geography, expected access frequency, perceived bandwidth, and server software. Expected access frequency is, where possible, determined from hit counters, or alternatively from the type and content of the site. For example, Web sites hosting small businesses are likely to be less heavily loaded than those of a large ISP. The downloaded resources are video or archive files (AVI, MPEG, ZIP) between one and two megabytes in size. The common occurrence of these file types and the fact that they tend to be large makes it easier to locate appropriately sized resources over a diversity of server types. Our chosen resources are larger than in the Zeus experiment, since our study of dynamic rate fluctuation required that downloads be lengthy.

Our client software (source is available [50]) downloads all twelve resources in sequence, then repeats, continuing until a 24-hour period had elapsed. Typically, this gives rise to a sample frequency for each target host of around 10 minutes for the 10Mbit client, and 20 minutes for the 0.5Mbit client. However, this varies given changing network conditions throughout the day, sometimes rising to 30 minutes and 50 minutes respectively during periods of exceptionally high load. For each transfer, the client records failure, connection latency, pattern of dynamic rate fluctuation, and average transfer rate. In the following sections we deal with each of these in turn.

Failure

Although our software implemented a timeout on downloads, none occurred. This is because the underlying Java socket implementations implement their own timeout on *socket activity* that overrides our lengthier timeout on *entire download*. Because of the Java socket timeouts, we cannot distinguish between latency timeout and other connection errors such as connection refused, for example. Similarly, socket errors are either manifestly terminated sockets mid-transfer or timeout by the Java socket implementation and we cannot distinguish between them. However, we can distinguish between connection failures and mid-transfer failures, and the following table shows the percentage of failure types for each target host across all of our experiments.

		T1 Fail %		Cable Fail %	
Server	**Locality**	**Connect**	**Socket**	**Connect**	**Socket**
g2301m.unileoben.ac.at	Austria	-	-	2.4	-
members.aol.com	E. USA	-	-	-	-
web.staffs.ac.uk	UK	-	-	0.6	1.1
kelim.jct.ac.il	Israel	1.2	-	5.1	-
www.auburn.edu	E. USA	**10.2**	-	**9.9**	0.5
www.arch.su.edu.au	Australia	1.7	-	1.6	-
www.jpweb.co.jp	Japan	2.0	-	-	0.6
www.fast.co.za	S. Africa	0.8	-	0.7	-
liv.auriga.ru	Lithuania	2.4	1.9	2.8	0.8
canyonsw.pair.com	W. USA	1.2	-	-	-
www.royalmail.co.uk	UK	-	-	-	-
www2.cristorei.com.br	Brazil	-	-	1.3	-

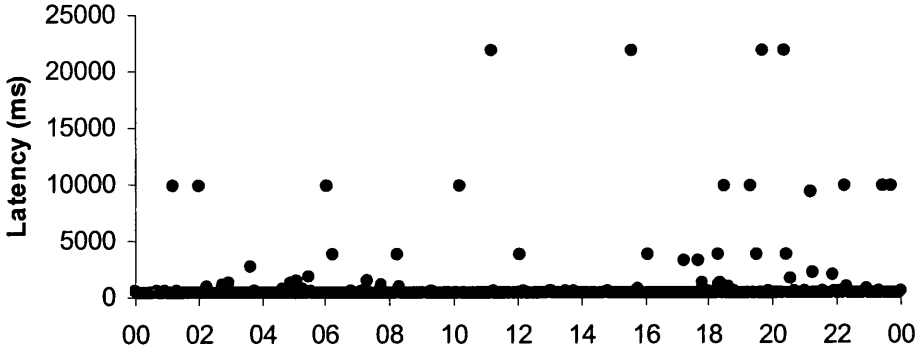
In a study of failure such as this, the amount of continuous time spent observing the target servers is important. The longer the time, the better the reflection of a particular servers' true failure characteristics. Our study pales in comparison to the Zeus study, since our testing time was approximately 48 hours for each server. However, our study incorporates a geographically diverse set of servers, unlike the Zeus study.

The data suggests that failure is slightly more common with geographically distant sites, though because our experiments were over a short time scale, we cannot draw concrete conclusions. The particularly high failure rate for www.auburn.edu is primarily due to the fact that the server failed overnight during one of our tests and coincidentally both T1 and cable modem experiments were running simultaneously.

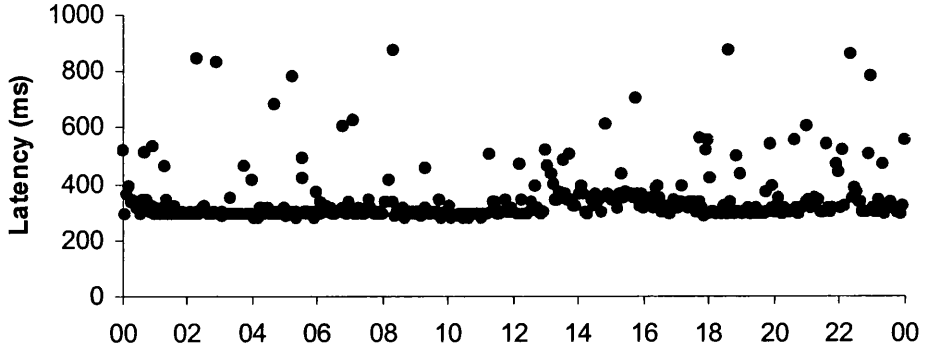
Latency

Our measurement of latency is the time taken to form a socket connection with the server, send it the request, and receive the resource http header information. In the set of graphs that follow, we give a representative sample of how connection latency is affected by time of day.

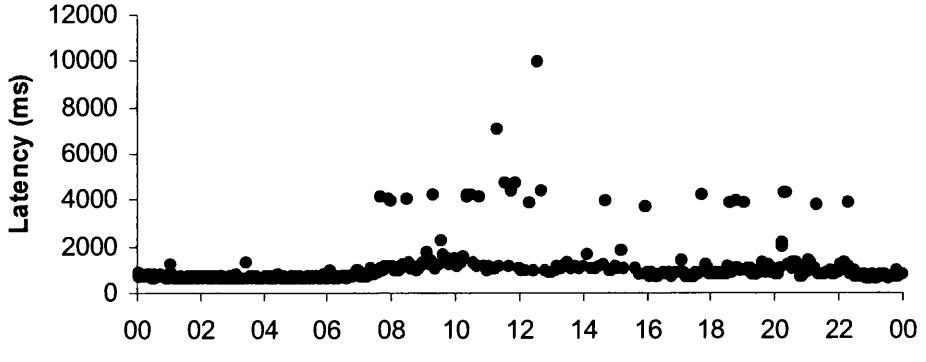
www.auburn.edu



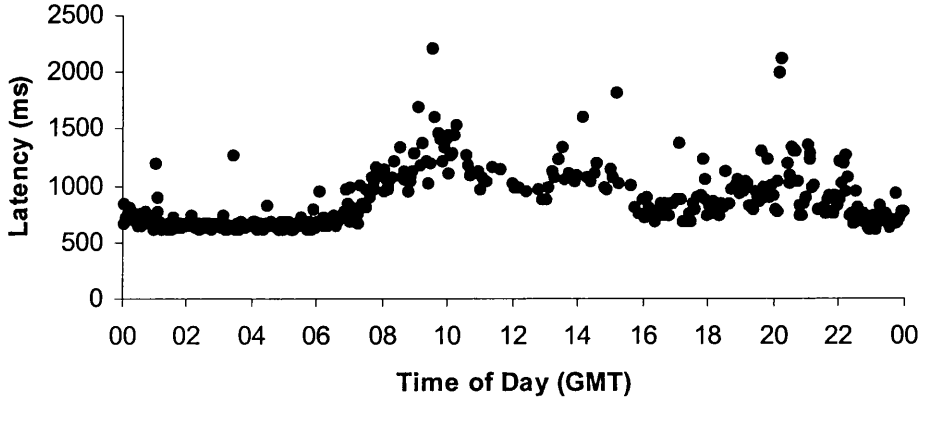
www.auburn.edu (zoom)



kelim.jct.ac.il

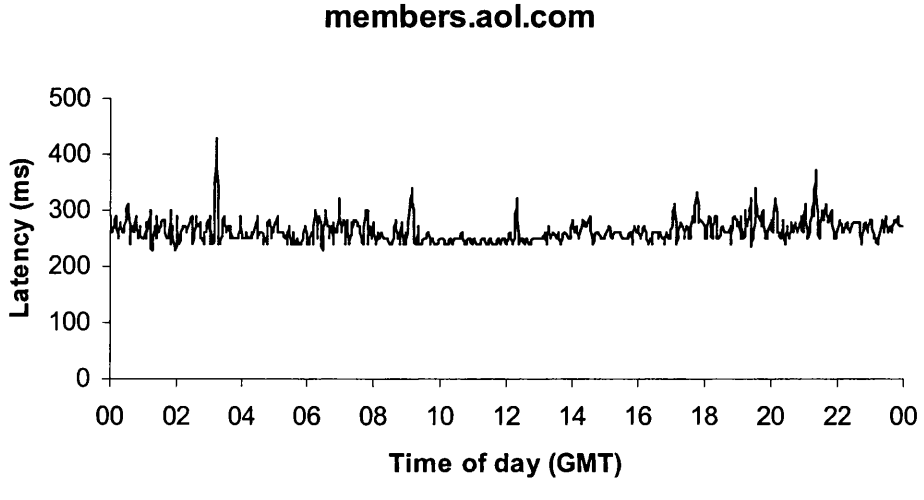


kelim.jct.ac.il (zoom)



In some servers, there is a marked increase in latency during periods of network congestion, and in others there is very little. For both servers shown above we chose to use a plot rather than line graph and show zoomed regions, because periodically, queries experienced exceptionally high latency. Contrast this with the graph below. The AOL server resides in the Eastern USA, and the USA is between five and nine hours behind GMT. This means that the consistent period in the centre of the graph corresponds to the early morning hours in the USA.

The pattern of exceptionally high latencies is repeated in all of our test servers, with the exception of the AOL server running iPlanet, where it is totally absent (see graph below). This indicates that there might be a problem in Apache, Microsoft, and NCSA servers whereby intermittently, connection attempts are not responded to for long periods. For example, the median connection time for the auburn server in the graph above is approximately 0.3 seconds. However, 13 of the 467 connection attempts (nearly 3%) took upwards of 10 seconds. There is also distinct 'banding' of latencies at 10 and 20 seconds. Some of these delays occurred at times where server load was low. We know this because the latency anomalies are not related to time of day, transfer rate of these long latency connections was normal, and fluctuation in rate was also normal (we discuss rate and variability in later sections). Thus, the intermittent high latencies seem to be an artefact of the servers. The fact that the iPlanet server does not seem to suffer from the anomaly at all supports our conclusion that it is servers and not the network to blame.



Although there might be problems with these servers, their behaviour does not break any semantic rules. That is, their behaviour still falls within what is valid behaviour for a server, as defined by http. There is nothing in the http 1.1 specification that states a requirement for timely server response. The only area where timeliness is addressed by the http protocol is

with error code 503, *service unavailable*. According to the protocol, this should be returned when the server is "temporarily unable to handle the request due to overloading". The server can optionally return a value in the header to indicate how long the client should wait before retrying the request. However, The http specification offers no definition of what constitutes overloading. That is, there is no specification of how long requests should be allowed to queue at the server.

An ongoing study by Netcraft [51] shows that 62% of all Web sites currently run Apache and 20% run Microsoft IIS. Thus, this latency anomaly is a tangible world-wide phenomenon that probably deserves more investigation. It certainly affects failure interpretation. For example, in a 48-hour period, 1.8% of transfers from www.jpweb.co.jp had an anomalous latency of over 12 seconds. In the same period the server suffered 2% *actual* connection failures. A Web application might interpret failure on a connection timeout of 10 seconds for a server, the assumption being that if the server has not responded by this time it has probably crashed. This might be reasonable for www.jpweb.co.jp, since its median latency is less than a second. However, because of the latency anomaly, failure would be interpreted wrongly approximately 50% of the time.

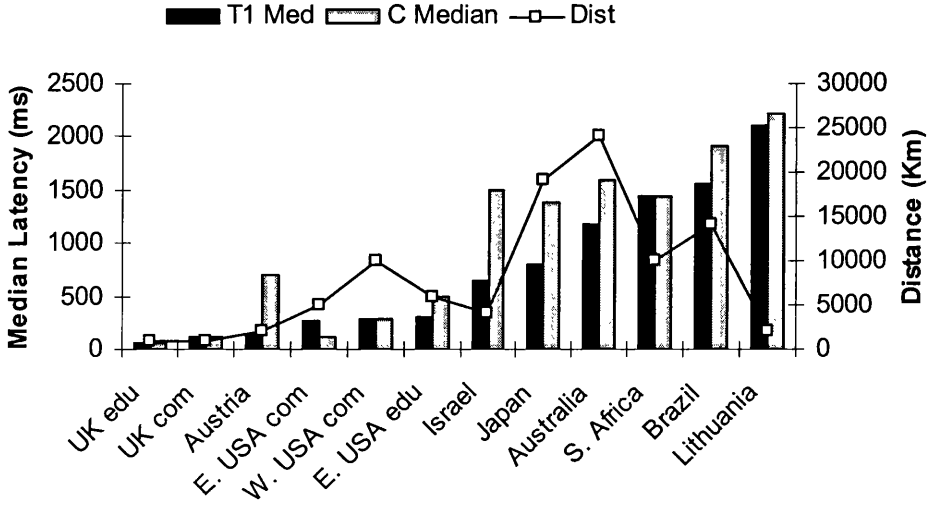
In the table below, we present our experimental results concerning latency. We measured latency in milliseconds, and show both mean and median latency. A large discrepancy between mean and median suggests that the server suffered from the latency anomaly. The table also shows the type of server software, number of network hops to the target, geographical distance (km), and the time taken for light to travel that distance (ms).

Geographical distance is approximate, rounded up to the nearest thousand kilometres. Note that geographical distance is not necessary a direct path across the globe. Instead, we model 'wire length'; calculated by observing the route that packets take through the global network topology. Australia, Brazil, and Japan are routed via the USA, and Lithuania is routed via Sweden. Although IP packets are routed dynamically, we observed that in general the same route is followed for groups of packets sent in a short time scale. For many sets of connections, we checked the route before and after, and did not find any major discrepancies that might significantly compromise our calculation of geographical distance or hop count.

Light speed in a vacuum is 300,000Km/sec. We use this to calculate the minimum possible latency for connections between the UK and the target host. Note that we doubled the light travel time calculated from distance in order to represent the fact that connection latency corresponds to a round trip.

					T1 Latency		Cable Latency	
Locality	**Server**	**Hops**	**Dist**	**Light**	**Medn**	**Mean**	**Medn**	**Mean**
Austria	Apache	16	2000	20	141	188	710	5633
E. USA com	iPlanet [52]	21	5000	40	260	262	110	160
UK edu	Apache	13	1000	20	50	68	60	87
Israel	Apache	18	4000	20	650	786	1490	2040
E. USA edu	Apache	26	6000	40	301	820	500	1025
Australia	Apache	30	24000	160	1181	1844	1600	3055
Japan	Apache	30	19000	120	791	1144	1380	1540
S. Africa	NCSA	24	10000	60	1442	1959	1430	1900
Lithuania	Unknown	21	2000	20	2098	3002	2210	3127
W. USA com	Apache	19	10000	60	280	592	280	562
UK com	Microsoft	13	1000	20	110	243	110	122
Brazil	Microsoft	20	14000	100	1552	2278	1920	2396

Taking the ratio of median latency to hops, we see little consistency across servers. This means that the number of hops is probably not the primary factor in determining latency. Likewise, we cannot correlate latency and distance. Consider the graph below, which indicates that Lithuania, South Africa, and Brazil buck the trend.



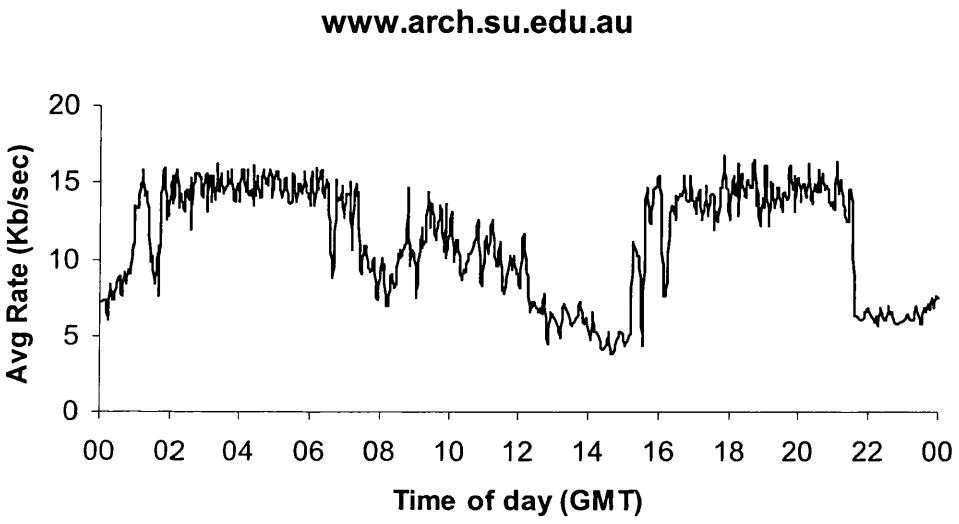
We avoid any speculation about possible increase in latencies due to the nature of network infrastructure in these countries. However, despite the high latency with respect to distance and hop count for these servers, there is an obvious crude relationship in that larger distances and hop counts result in longer latency. Overall though, the relationship between hop-count, distance, and network and server congestion that determines latency must be a complex one.

In short, it is difficult to draw many conclusions from our studies of network latency, other than that networks are significantly slower than light, and that latency is crudely proportional to hop count and geographical distance. We suspect that server load plays a major role in determining connection latency. However, it is difficult to introduce control experiments in this regard. In principle, an artificial situation could be constructed to test the impact of server load on connection latency, but such an experiment would require a level of time and resources that puts it beyond the scope of this thesis.

One observation we can make about latency is that for geographically distant servers latency often remains consistent over time at the lowest latency bound. In general, this consistency occurs during periods of low server and network congestion. Any variability arising from network and server congestion will be 'upwards'. The graph above for the Israeli server is a good example of this behaviour. This indicates that there is a fundamental lower bound on latency that irrespective of network conditions cannot be improved upon given the same hardware. This suggests that the fundamental light-speed barrier is important in determining the lower bound of latency for global communication and that geographical distance and hop count factors are not responsible for variability in latency.

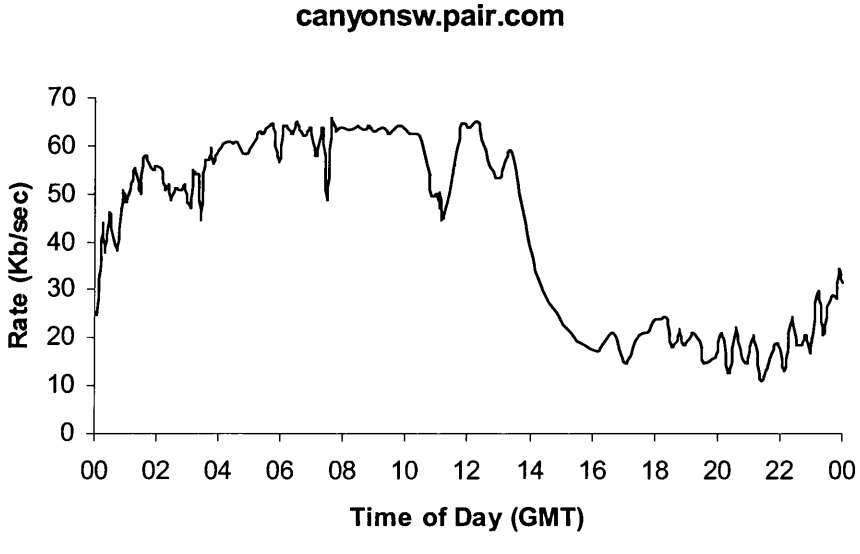
Average Rate

Transfer time is heavily dependent on the size of the resource being downloaded, so we choose transfer rate and latency as our performance measurements. Intuitively, one might think that transfer rate is independent of the size of the resource being downloaded. However, some Web servers, Apache included, can be configured by the site administrator to prioritise downloads according to the length of time they are taking. For example, a server might

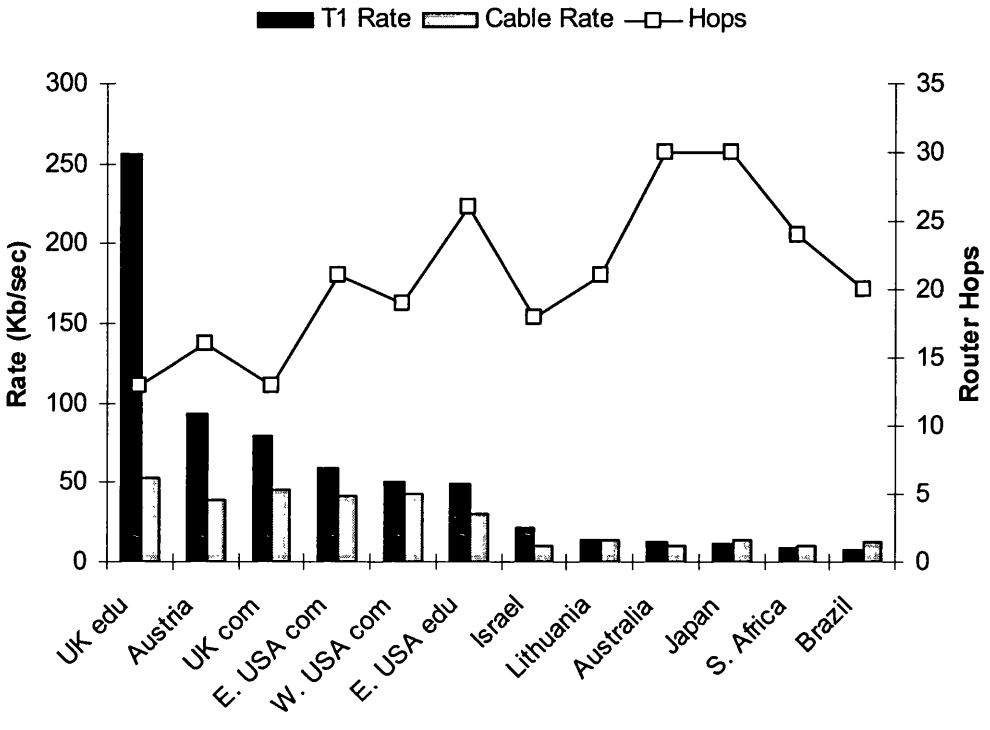


gradually reduce the priority of long downloads over time, resulting in a trend towards lower transfer rate that impacts the overall average transfer rate. There is nothing at the http protocol level that allows determination of such scheduling policies, and rate trends may not be distinguishable as policy. It is perhaps to be expected that there can never be a perfect control group for rate experiments, since heterogeneity is an intrinsic property of the Web. However, we attempt to minimise potential anomalies in our results by choosing a diverse range of servers, and importantly, remote resources of approximately the same size (between one and two Mb). We are particularly interested in how average transfer rate varies with network and server load. By calculating the average transfer rate for a series of downloads across a 24-hour period, we can expose trends in rate for network and server congestion related to the time of day.

The www.arch.su.edu graph shows the trends in average rate for the test server in Sydney Australia against time of day. There is twelve hours of time difference between server and client. Because the server is academic, we expect network rather than server load to be the primary factor affecting transfer rate. We can see that average rate is low during the working day in Europe, falls even lower as the work day begins in the USA, and drops sharply in the late evening, as the work day begins in Australia. This suggests that network traffic between the UK and Australia is routed either through the USA, or by satellite. In this particular case, tracing packet route shows that the connection is via the USA. Peak rate is around 15K, and troughs at around 4K at the point of highest load, which is a 70% reduction in performance. Although rate varies throughout the day, the fact that transfers at approximately the same time achieve similar average rates suggests that rate is consistent given similar network and server load. The canyonsw.pair.com graph shows trends in average rate for a server in California, which is eight hours behind UK time. Transfer rate from this server seems largely unaffected by the start of the workday in the UK, but falls by 70% during the workday in California.



The following graph and table shows our results for average rate over a 48-hour period. Broadly, average transfer rate is inversely proportional to hop count.



			Rate (Kb/sec)	
Locality	**Hops**	**Dist (km)**	**T1**	**Cable**
Austria	16	2000	93	39
East USA Com	21	5000	59	41
UK Edu	**13**	1000	**256**	**53**
Israel	18	4000	21	10
East USA Edu	26	6000	49	30
Australia	30	**24000**	12	10
Japan	**31**	19000	11	14
S. Africa	24	10000	9	**10**
Lithuania	21	2000	14	14
West USA Com	19	10000	50	43
UK Com	**13**	1000	80	45
Brazil	20	14000	**8**	13

Dynamic Rate Fluctuation

Transfer rate is calculated by dividing a number of bytes transferred by the amount of time taken to transfer them. Thus the rate of transfer for an entire Web fetch is the size in bytes of the resource divided by the time taken to download it, after subtracting connection latency time. However, we are interested not only in the transfer rate for the entire resource, but in how transfer rate fluctuates across the duration of a Web fetch. Transfer rate at a particular moment in time is essentially meaningless, since no bytes can be transferred in zero time. Instead, we must break down the overall transfer into sampling periods over which we calculate the rate. There are two pragmatic issues relating to these sampling 'windows'. Both derive from the fact that language APIs tend to provide little control over how bytes are read from a socket stream.

The low-level implementation of a Web fetch requires repeatedly reading from a socket until the data stream is exhausted. In both the C and Java implementations of sockets, we can specify a maximum number of bytes to return from a read operation. However, the actual number of bytes read can be less than that, and can even be zero. Moreover, the time taken to complete each read operation is non-deterministic, though is probably upper-bounded internally. In short, this means that the time for each sampling window cannot be fixed at this level. The end result is that we have data for the number of bytes transferred at a series of increasing times, but samples are at inconsistent intervals.

Our solution is to keep track of elapsed time and enforce a minimum sample window time, by repeatedly invoking read operations and counting bytes until the minimum time has passed. Then we can approximate rate sampling at regular discrete intervals by linear interpolation. We calculate the rate for each inconsistent interval by dividing the number of bytes transferred since the end of the last interval by the length of that interval. Then, given the two values r_a and r_b for the rate over the interval bounded by times t_a and t_b respectively, the transfer rate r at the i^{th} sample point (lying between a and b) is

$$r = m\, t_i + b$$

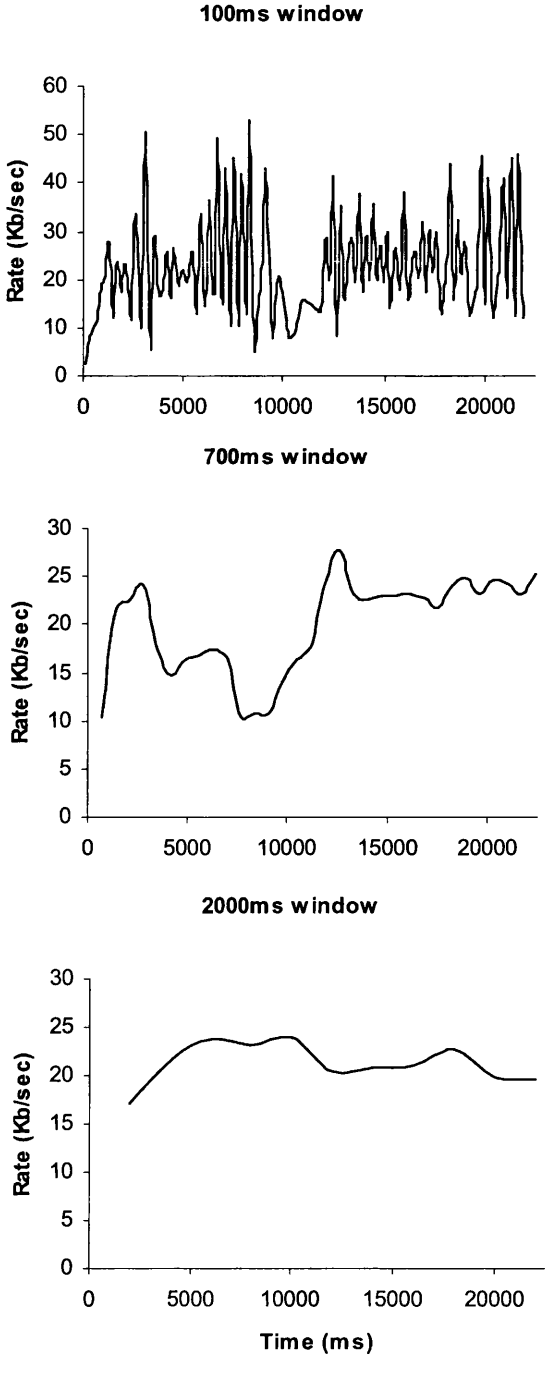
where m is the gradient

$$m = (r_b - r_a) / (t_b - t_a) \text{ and } b = ra - (m\, t_a)$$

That is, we calculate rate at the i^{th} discrete window interval by calculating the gradient and offset of the line passing through the nearest actual samples on either side of the desired time. This allows us to calculate the rate at any point in between. There are other methods of interpolation, such as polynomial and cubic interpolation. These use data points additional to the two on either side of the target and find a curve that passes through them all. The curve

equation can then be solved for rate at a required time. This can give more accurate results if the sample points are widely spaced, but we found that in many cases the nature of the curve fitting would result in negative interpolated rate values, which are unacceptable.

The second pragmatic issue relating to sampling windows is granularity. That is, how long the sampling window should be. If the window is large, then there is a possibility that significant fluctuations in rate might be masked. For example, rate might be at 10K, fall to zero, then rise to 20K all within five seconds. If the window were five seconds long, the rate observed is 10K, masking the fall to zero and peak at twenty. In contrast, a window of 100 milliseconds might at times indicate a rate of zero, whereas throughput is actually quite high at, say, 20K. Window size should minimise variability in the very short-term, while retaining an accurate picture of rate fluctuation overall.

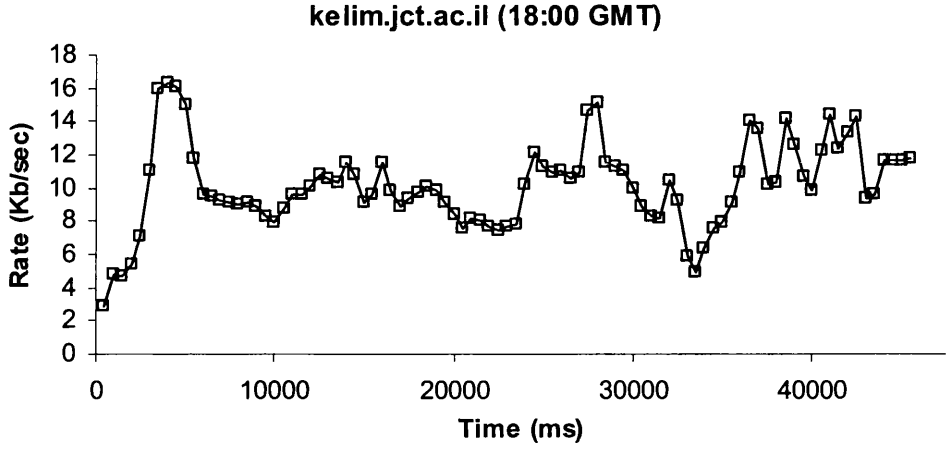


The choice of window size is essentially qualitative, but we performed an experiment, downloading several resources at different times of the day, with different window sizes. For small windows, the accuracy of the timer is an issue. Our language of implementation is Java, and we found that the Java system clock is not accurate below a granularity of around 30ms. Thus, we choose a minimum window size of 100ms, ranging up to two seconds. Consider the three graphs adjacent, which reflect typical results for small, medium, and large window sizes. All graphs correspond to fetches of the same resource from a single server under similar network conditions. We cannot show three graphs for precisely the same fetch, since only one window size can be used for any given fetch. However, we performed experiments indicating that several transfer of the same

resource within a short time scale gave rise to similar patterns of fluctuation.

A window size of one hundred milliseconds gives rise to a 'stepped' effect and jaggedness in rate fluctuation, and the window size of two seconds potentially misses significant troughs and peaks. A window size of 700 milliseconds is a compromise that appropriately reflects fluctuation in dynamic rate.

On examining the trends in perceived bandwidth (actual transfer rate), we see that it fluctuates unpredictably throughout transfer, though fetches generally start slowly. Consider the following graph, which represents the rate fluctuation of a transfer from Israel at 18:00GMT. The pattern of fluctuation is representative of typical rate fluctuations across the duration of different fetches during average network congestion.



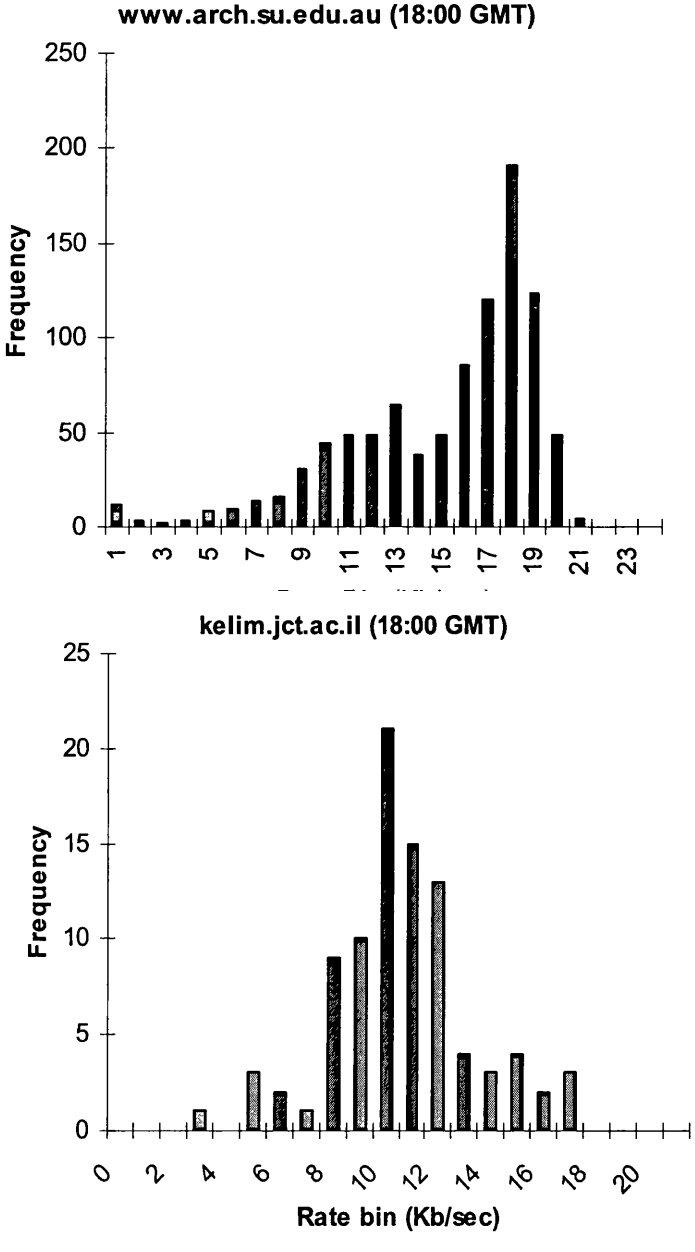
We are interested in the degree to which transfer rate throughout a fetch fluctuates in relation to the time of day (and thus network and server load). To determine the level of fluctuation, we consider the spread of rate observations about the mean. The larger the spread, the more fluctuation. One way to calculate the spread of a data population is by calculating the standard deviation, *SD*. In the following equation, n is the size of the population and x is a member of the population.

$$SD = \sqrt{(\Sigma\, x^2 - (\Sigma\, x)^2 / n(n-1))}$$

The theoretical basis of standard deviation is complex and beyond the scope of this thesis. However, one practical concern is that the population from which the data arises should be a distribution that is approximately Gaussian. Gaussian distributions are represented by a family of curves that are defined uniquely by two parameters: the mean and the standard deviation of the population. Gaussian curves are always symmetrically bell shaped, and the extent to which the bell is compressed or flattened out is related to the standard deviation of the population.

The two graphs below show histograms for rate. The graph for kelim.jct.ac.il corresponds to the transfer rate graph shown above. On the histograms, frequency is the number of rate observations that fell within that 1Kb/sec range, or bin.

These kind of distributions are representative of the vast majority of our experimental results. For the upper histogram, there is a tendency for samples to be of higher rate. Many transfers follow this pattern, which corresponds to the situation where rate is broadly consistent, but periodically troughs. The fact that there are more troughs than spikes causes the skew. When interpreting failure based on rate, we are not interested in spikes, only troughs. In the lower histogram on the right, individual troughs represent a larger deviation from the mean than the troughs of the Israel histogram.

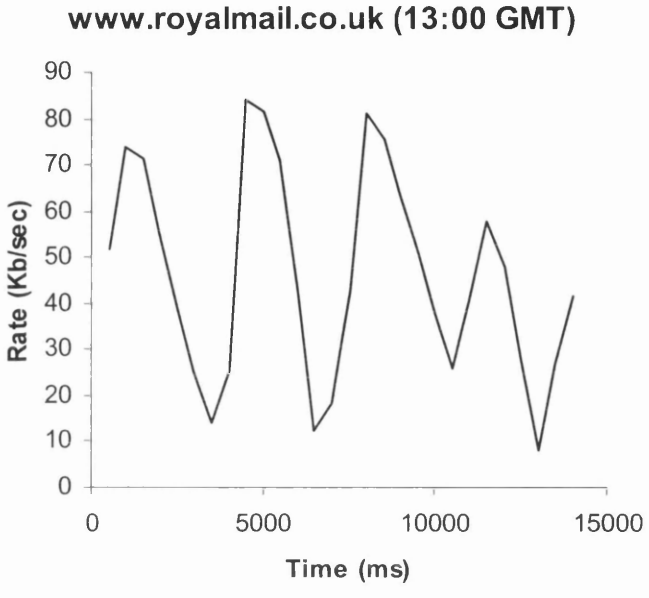


Now, skewed distributions (not Gaussian) tend to inflate standard deviation. However, since it is troughs that are relevant for interpretation of failure, and in particular may result in erroneous interpretation of failure when overall transfer progress is good, the larger standard deviation serves as a cautionary indicator in this respect. The higher the standard deviation, the more variable transfer rate. An inflated standard deviation reflects the fact that a transfer suffers from many troughs, which are the most insidious form of rate variability. Thus, although we draw the line at saying non-Gaussian distributions are a positive benefit, we feel that the implications do not impinge on any conclusions we draw from studies of standard deviation.

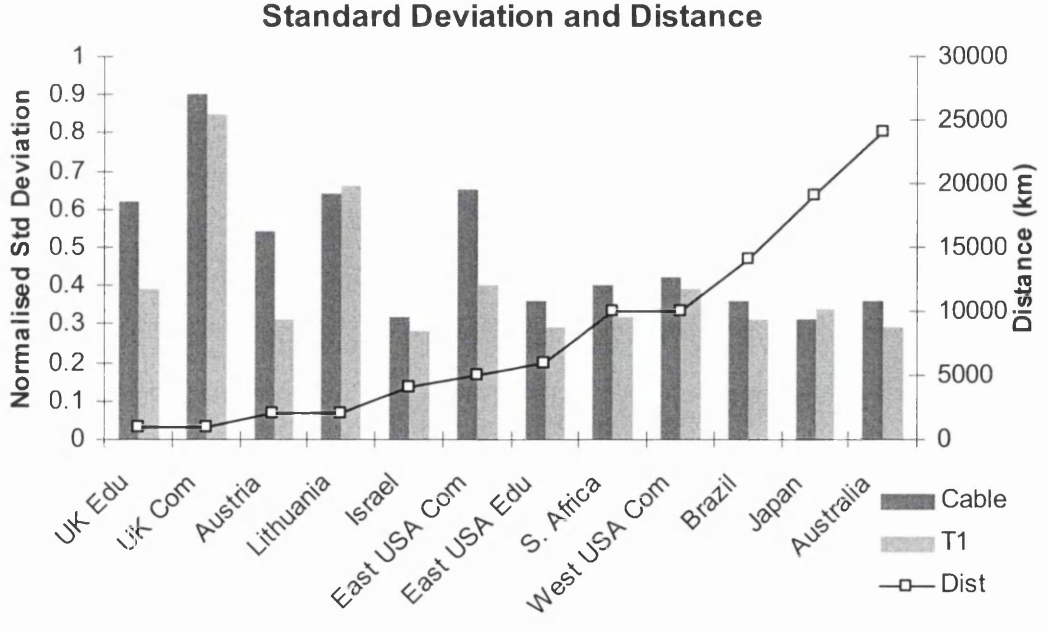
The following table contains figures for each target site that represent the average *normalised* standard deviation in rate (higher numbers represent increased variability) for Web fetches sampled over 24 hours. We normalise by dividing the average standard deviation of all transfers by the average download rate for all transfers. The value then reflects deviation from average rate in fractional terms that allows us to compare the variability of sites with different perceived bandwidth. Interestingly, we can conclude that rate fluctuation does not appear to be a function of geographical distance and so not a function of the number of intermediate network nodes.

		T1		**Cable**	
Locality	**Dist**	**Mean Rate**	**Std Dev**	**Mean Rate**	**Std Dev**
Austria	2000	93	0.31	39	0.54
East USA Com	5000	59	0.40	41	0.65
UK Edu	1000	**256**	0.39	53	0.62
Israel	4000	21	0.28	10	0.32
East USA Edu	6000	49	0.29	30	0.36
Australia	**24000**	12	0.29	10	0.36
Japan	19000	11	0.34	14	0.31
S. Africa	10000	9	0.32	10	0.40
Lithuania	2000	14	0.66	14	0.64
West USA Com	10000	50	0.39	43	0.42
UK Com	1000	80	**0.85**	45	**0.90**
Brazil	14000	8	0.31	13	0.36

Surprisingly, it is a UK server that exhibits the highest variability in rate, for both T1 and cable connections. Adjacent, we show a graph of a typical pattern of dynamic rate fluctuation for this server. Note that high standard deviation is to be expected if the mean is a large value. However, we have normalised by dividing by the mean in each case, which allows us to meaningfully compare the variability of different populations.



The bar graph below shows the same data as in the table, but in a format from which we can easier draw conclusions. There is a suggestion that standard deviation is inversely proportional to geographical distance. However, it is more likely that it is the fact that closer servers tend to have higher transfer rates, and it is higher transfer rates that give rise to variability.

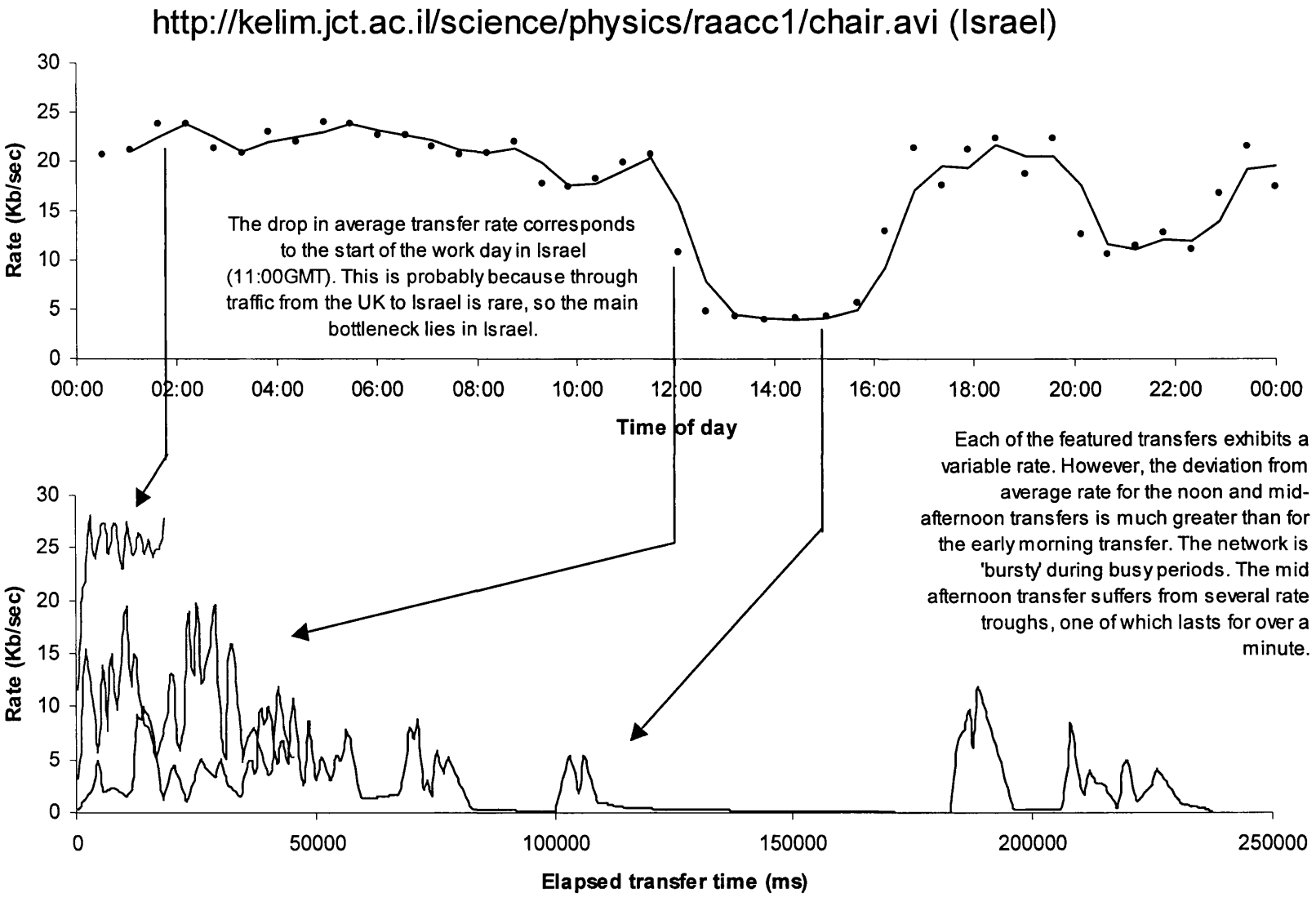


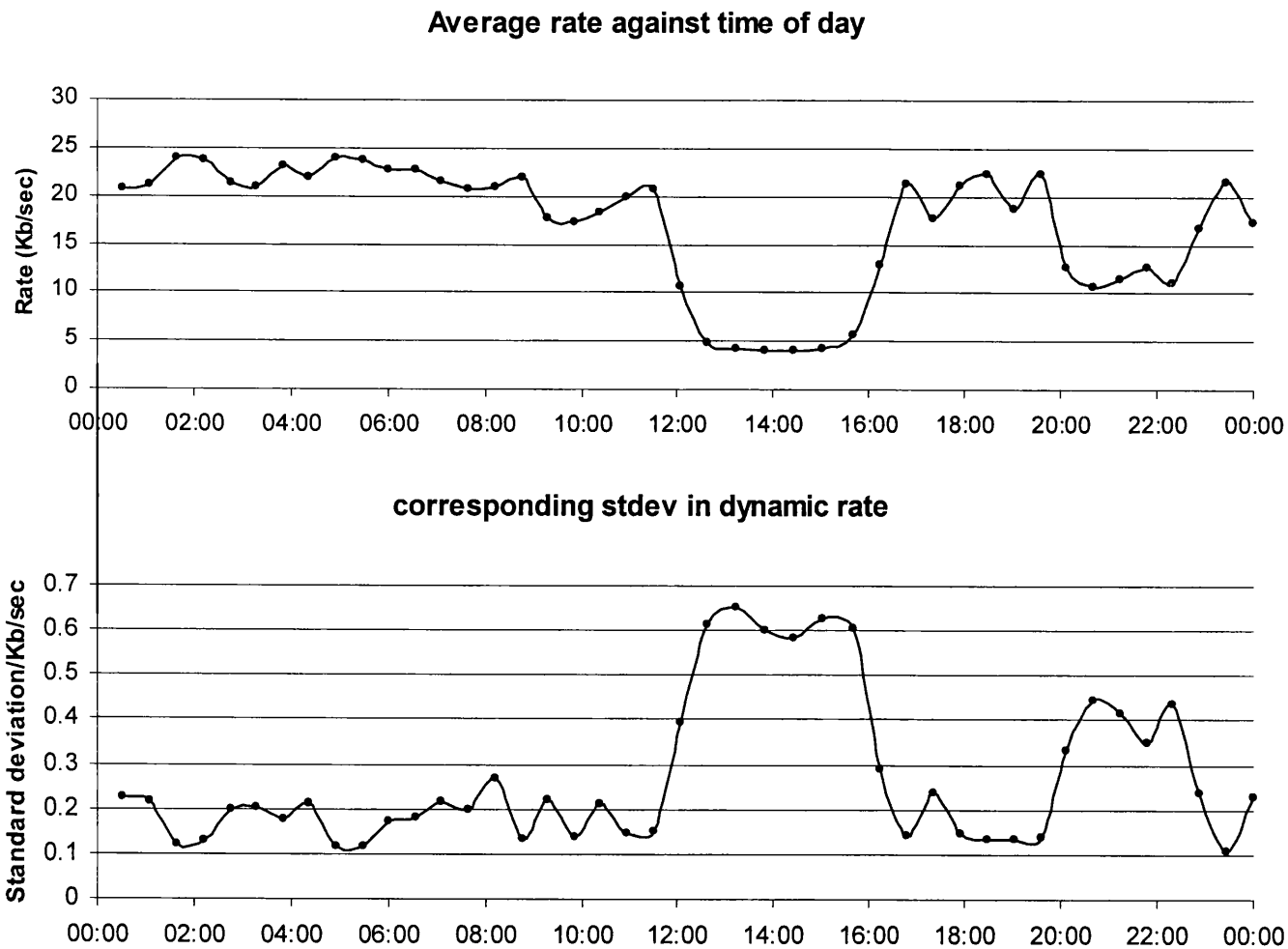
For T1 connectivity, the AOL server also has a high average variability, and like the Royal Mail site is likely to be heavily loaded. Server loading may be a significant factor in determining rate variability in addition to network load. Both are functions of the time of day, but we distinguish between variability introduced by network load and by server load by

comparing the standard deviation between servers that co-located in network topology but are known to have different access frequency. We can see that from the table, that East USA Com (AOL) and East USA Edu (Auburn) are co-located, but have different standard deviation in average rate. The site with higher expected server load (AOL) has higher rate variability.

Closely following AOL in T1 variability is the UK academic site. Although this site has the highest average bandwidth, overall that bandwidth is extremely variable. Examination of a number of dynamic rate graphs for this site shows that the morning is the most variable time, and troughs are frequent.

On the following two pages, we present some graphs and analysis for a particular set of transfers from the Israel test server. The first pair of graphs show how rate fluctuation is affected by network and server load. The second pair shows that there is a correspondence between variability in rate and time of day.


http://kelim.jct.ac.il/science/physics/raacc1/chair.avi (Israel)
Rate (Kb/sec)
30
25
20
15
10
5
0
00:00
02:00
04:00
06:00
08:00
10:00
12:00
14:00
16:00
18:00
20:00
22:00
00:00
Time of day
The drop in average transfer rate corresponds to the start of the work day in Israel (11:00GMT). This is probably because through traffic from the UK to Israel is rare, so the main bottleneck lies in Israel.
Each of the featured transfers exhibits a variable rate. However, the deviation from average rate for the noon and mid-afternoon transfers is much greater than for the early morning transfer. The network is 'bursty' during busy periods. The mid afternoon transfer suffers from several rate troughs, one of which lasts for over a minute.
Rate (Kb/sec)
30
25
20
15
10
5
0
0
50000
100000
150000
200000
250000
Elapsed transfer time (ms)



The upper graph is the same as on the previous page. Data points in the lower graph represent the corresponding standard deviation for rate across transfer duration, normalised against the average rate. That is, we take the standard deviation for rate samples and then divide by the average rate, thus allowing the standard deviations to be compared on the same scale. In practical terms this graph reflects the level of variability in transfer rate. High values correspond to highly variable transfer rates, and low values to consistent rates. There is a clear correspondence between rate, variability, and time of day. We examined standard deviation in this way for several different sites, and all showed similar correspondence, but to varying degrees.

This shows that in addition to decreasing, transfer rate fluctuates more when the network is under heavy load. That is, data transfer is more 'bursty'. Furthermore, rate 'troughs' are common, even when the network is not under particularly heavy load. However, troughs are more frequent, and of longer duration when the network is heavily loaded. Troughs can range in duration from several seconds to several minutes. The fact that there is a clear correspondence between time of day and standard deviation in rate shows that network and server load, and not geographical distance, are the primary factors in determining rate variability during Web fetches.

Conclusions

Our experiments allow us to draw several conclusions that may be of use when programming Web applications. To summarise, for each conclusion we speculate how they might relate to programming Web fetches.

- *Median latency is a function of geographical distance* – if the geography of a site is known to be distant, then programmers may wish to extend any latency timeout over that normally applied to co-located sites.
- *Latency is intermittently very high* – an anomaly we suspect to be related to server type causes exceptionally high latencies to be observed periodically. This anomaly can cause the majority of failure interpretations by latency timeout to be erroneous. An appropriate response to this might be to immediately retry all connections that timeout on latency.
- *Perceived bandwidth is inversely proportional to distance* – as with latency, programs should expect geographically distant servers to experience lesser performance than local servers. For example, a transfer rate of 4Kb/sec to a site in the UK is poor and might justify termination. In contrast, for a site in Australia this transfer rate should probably be deemed acceptable, unless an equivalent resource can be found on a local server.
- *Perceived bandwidth is affected by network and server load* – during known busy periods of the day, programs should expect to observe a reduction in perceived bandwidth. Programmers may wish to relax performance expectations during these periods.
- *Bandwidth fluctuates throughout transfer* – fluctuation in transfer rate is more marked than the transfer rates shown by browser applications would have us believe. When automating failure interpretation, programmers should remember that rate can drop to zero for short periods, without indicating that overall transfer progress is poor.
- *Bandwidth is more variable during periods of network congestion* – in addition to the previous point, programmers should take extra care not to inadvertently interpret failure for a connection based on rate troughs during known periods of network congestion. We return to the issue of diminishing the impact of rate troughs in the context of calculating dynamic rate in Chapter 5 – *Persistent Relative Observation*.
- *Bandwidth variability is not a function of geographical distance* – since variability is primarily a function of server and network load, programmers can ignore server locality as a factor when rate troughs may be an issue.

- *Average rate for distinct transfers is consistent in the short term* – rate shows definite trends according to the time of day but is unlikely to change much from one minute to the next. This means that programmers should not expect to see marked differentiation in the transfer rate of successive or concurrent fetches to the same server. This property is important when we later develop a technique for failure interpretation based on *relative observation.*

- *Rate troughs are common even under moderate network load* – troughs, where transfer rate intermittently drops to zero, can range in duration from several seconds to several minutes. However, they are more frequent and of longer duration when the network or server is heavily loaded. Programmers should understand that rate may drop to zero for extended periods when the network is congested. Thus, programs should interpret failure by rate observation less readily when under these conditions.

Now that we have examined the nature of the Web domain in quantitative terms, we can go on to examine the models that programming languages employ to detect failure and direct flow control after the fact. In particular, we are interested in how the domain properties are mapped into programming languages, and in the appropriateness of the flow control mechanisms.

3: Domain Properties and Flow Control

To program in the Web domain, we need something in the semantic space that reflects what is going on at the communication level. A language designed specifically for programming Web applications should provide fundamental operations for Web access. It is the semantics of these operations that expose the nature of the Web domain. Since Web access is prone to failure, the primitive access operations should be integrated with an appropriate flow control abstraction for failure. In turn, this flow control mechanism should allow the expression of concurrency. Before we continue, we define some terminology.

A specification of *what constitutes failure* is the first part of a computation's *failure semantics*. The second part is the specification of *flow control after failure*. In other words, a computation's failure semantics specify the meaning of failure: defining how a computation fails, and what to do if it does. *Failure representation* can be a simple or complex value, or even a procedure-like entity. More discussion of failure representation can be found in Chapter 8 – *Exception Handling*. A language's *failure model* defines the failure representation and the means to program failure semantics (how failure detection can be expressed, and the possible flows of control that can be expressed). A particular failure semantics for a computation is an instance of what is programmable within the language's failure model.

Overloading flow control for failure onto function return

During program execution, it is sometimes necessary to determine the existence of exceptional circumstances that require special processing. In particular, this applies to failures that can only be detected dynamically. In general, failure to complete an operation is detected by the operation itself, but the significance of that failure is known only by the operation's invoker, since only the invoker knows to what use the operations results are being put. Thus, on the detection of failure, information about the failure must be passed to a higher level of abstraction so that appropriate remedial action can be taken. For example, a programming abstraction for network IO cannot determine the significance of failure to make a connection. Only its invoker can, and so information about the failure must be propagated up the dynamic invocation chain. If the detector of failure can determine the significance of that failure, then it should not be considered an exceptional circumstance and should instead be handled with normal flow control by the operation that detects it.

A programming language or operating system may intervene on the programmer's behalf to detect dynamic errors such as out of bounds array indexing or division by zero, for example. For other errors, programmers write explicit error tests within a function that may cause it to return an error code instead of a result, possibly with additional error information in global

state. The function's invoker can then examine the return code and deal with the error in the context of knowing its significance. Although this is the most common way to indicate the presence of errors, there are others, and Levin provides a detailed examination of the possibilities [53].

Perhaps the most well known example of the function return overloading methodology is that associated with the C programming language. Many C standard library functions return integers that encode information about their execution. For example, an IO function might return a positive integer that indicates the number of bytes read, or a negative integer indicating the occurrence of an error, the type of which can be determined from the value returned. Execution results that cannot be overloaded onto the return value are returned in reference (pointer) parameters. Consider the example below, written in C.

The readData function reads lines of text from a file into a buffer (passed as a reference parameter), separating them with a percent symbol. The first item of data in the file is the number of lines in that file. There are three main reasons why readData may fail, and these correspond to the three explicit error tests made:

- The file might not exist or cannot be opened.
- The first entry in the file might not be a number, indicating incorrect file type.
- The number of entries in the file might not correspond to the integer read.

Before the implementation of readData, we define integer codes that correspond to each of these possible errors. Within readData, we test the return values of the file IO function invocations, and if they indicate an error, we return the appropriate error code to the invoker of readData. The invoker tests the return value of readData in a similar manner. On encountering an error when opening the file with fopen, readData examines errno, which is a global variable set by many IO functions to carry information about failure in addition to that in the return code.

```
#define FILE_STRUCTURE_ERROR -1
#define READ_DATA_ERROR -2

int readData(char* fileName, char* buffer) {
        int numLines;
        int i;
        FILE* f = fopen(fileName,"r");
        if(f==NULL) {                                   /* error opening file */
                switch(errno)
                case FILE_NOT_FOUND: ...
                        return FILE_NOT_FOUND;
                ...
        }
        if(fscanf(f, "%d",&numLines) < 0) {             /* no line count error */
                ...
                return FILE_STRUCTURE_ERROR;
        }
        for(i = 0; i < numLines; i++) {
                int numRead;
                numRead=fgets(f,buffer);
                if(numRead<0) {                         /* file read error */
                        ...
                        return READ_DATA_ERROR;
                }
                buffer+=numRead;
                *buffer++='%';
        }
        return numLines;
}
...
switch(readData("file.txt",aBuffer)) {
        case FILE_NOT_FOUND : ...
        case FILE_STRUCTURE_ERROR : ...
        case READ_DATA_ERROR : ...
        default : ...
}
```

Overloading the function return mechanism in C can lead to function implementations that are difficult to understand, as in the example above. This is because error detection code and associated control flow code pervades computational logic, undermining structural coherence. Furthermore, use of this methodology cannot be captured in function interfaces. This means that the fact that a function may return an error code and the particular encodings used must be documented separately to the function signature. This complexity is compounded if side effect to global state also carries error information.

A separate problem related to the use of global state to carry failure information is that it precludes function *re-entrancy*, thereby compromising concurrency. Re-entrant functions can be executed simultaneously by two or more concurrent threads of computation, since they do not make use of global variables (or static variables, in C). If a function that is not re-entrant is executed in a concurrent context, two or more concurrent invocations might attempt to update the global state on which its execution depends. This can compromise the function's intended semantics.

With the function overloading methodology, error tests should be associated with every function invocation that can fail. However, this is not statically enforced so indolence or oversight on the part of the programmer can lead to errors going undetected. In some cases this can be catastrophic as errors enter the system unexpectedly and only manifest themselves some time later. Flater proposes [54] some extensions to C that eliminate many explicit tests of return codes for standard library functions, and for circumstances such as array bounds violation and null pointer dereference. However, the extensions do not address the issue of *propagating* error information, since the automatic behaviour on discovering an error code is to terminate the program. Gehani argues [55] that the function overloading methodology result in programs that are error-prone, lacking in modularity, have reduced readability, and are more difficult to reason about formally; concluding that it is inadequate for anything but the smallest applications.

Exception handling

Exception handling[1] is a particular abstraction for programming failure models that has been widely adopted in varying forms by both general purpose and domain specific programming languages. Essentially, all forms of exception handling mechanism amount to the same thing: automating the process of propagating error information to a higher level of abstraction that can handle it in the context of knowing its significance. Most mechanisms define a small set of system exceptions that are detected automatically by the language run-time, and can be

[1] We present a detailed survey of exception handling in chapter 8.

handled by the programmer in different ways depending on context. However, the real power of exception handling lies in providing the means for programmers to define and raise their own exceptions.

Programmers are responsible for implementing programming logic that detects the exceptional situation, and raises an appropriate exception. Thus, exception handling mechanisms do not directly address what constitutes failure or how failure can be detected apart from system exceptions. Thus, exception handling mechanisms are independent of any underlying conceptual domain such as Web observables, for example. Using exception handling to provide flow control for failure in the context of Web fetches requires the programmer to implement mechanisms for exposing the domain properties and failure detection based on these properties. Thus, in the context of the Web, exception-handling mechanisms are only two thirds of a failure model, since they provide only failure representation and the means to express flow control after failure. What constitutes failure must be programmed with general program logic, and the methodology applied may differ between programs, and even within the same program.

In general, exception handling mechanisms are based on the concept of absolute failure. In particular, automated support for failure detection in the form of system exceptions is purely concerned with absolute failures, such as out of bounds array indexing, for example. However, the investigation described in the previous chapter shows that it is perceived (or interpreted) failure that is more important in the Web domain. If we wish to develop failure models based on the perceived exception rather than the absolute exception, we require support to interpret failure based on dynamically available information. In general, exception handling mechanisms are not good at interpreting failure. Failure interpretation depends on hardware, the nature of the substrate (sockets etc) and other issues that we do not want to address explicitly in programs, since doing so requires implementing sometimes complex code that is incidental to computational logic. In principle, this could be provided by the programming system, so as to ease the programming of domain specific tasks, one approach is to bring domain concepts in at the language level.

In this chapter, we examine two direct approaches to Web programming language design: Cardelli and Davies' Service Combinator algebra [56] and the programming language WebL [57]. Both attempt to integrate exception handling with mechanisms that directly expose Web properties in the semantic domain, and aid in the detection of failure. These are mechanisms that are absent from general purpose exception handling. The Service Combinator algebra is a small formalism for specifying reliable Web fetches that is intended to be embedded within a general purpose language. WebL is a complete Web programming language based, in part, on service combinators. Throughout this chapter, we are interested primarily in how Service

Combinators and WebL expose the domain properties, while integrating mechanisms for failure interpretation, concurrency, and flow of control.

Service Combinators

Cardelli and Davies define a small formalism whose computational model is tailored to programming Web applications, based on the notions of *services* and *service combinators*. The service combinator algebra is intended to make the communication aspect of Web computations more reliable. It allows the construction of programs that when executed mimic typical human *web reflexes*, the 'algorithmic' behaviour exhibited by human browsers when attempting to retrieve resources on the Web, such as strategies for handling failure and slow transfer rates with timeout, retry, and concurrent download, for example. Although useful on its own, the designer's intention is for the combinator algebra to be integrated with a more general purpose functional programming language. One major benefit of the combinator algebra is that its simplicity and regularity facilitates formal reasoning about programs. Expressions in the algebra can be manipulated algebraically in interesting ways, and are amenable to simple proofs of program correctness.

A *service* is a high-level primitive that provides the information of a Web resource, and encapsulates error detection and handling. Since services correspond to Web server requests, when invoked they may fail to respond with information, or if they do, it might indicate failure. The information and error output of services can be composed in order to create new services with *service combinators*, potentially with the introduction of concurrency. The idea is to combine two or more similar services, which may be unreliable, in order to provide a more reliable 'virtual' service. Error recovery policy and concurrency are embedded within the combined service.

The algebra defines three main types of service: *url*, *get*, and *post*, which correspond to simple http document fetch, parameterised get, and post respectively. The latter two services may be passed any number of name/value pairs as parameters, and result in an http request equivalent in encoding to an html form submission. The observable properties of a service are its transfer rate in bytes per second, and its time since invocation in seconds. The algebra provides two combinators, *timeout* and *limit*, which allow the specification of failure interpretation for arbitrary services through monitoring of transfer time and rate. Applying timeout to a service results in a service that fails should its time since invocation exceed a given constraint. A limited service fails if any of its constituent services fall below a given constraint on transfer rate. The limit combinator also allows the specification of a startup time, over which the rate constraint is not enforced. The rationale here is that no service will begin receiving data immediately, so each limited service is given a period to connect before

failure can be inferred from transfer rate observation. Thus, the startup time is equivalent to a constraint on http request latency, since an unconnected service has a rate of zero.

Timeout and limit are combinators for specifying failure interpretation. The algebra provides three flow of control combinators for the purpose of specifying reliability. The *sequential* and *concurrent* combinators afford reliability through redundancy. Both are binary service operators, represented by the infix '|' and '?' operators respectively. The sequential combinator returns the result of either a primary service, or that of a secondary service should the primary fail for whatever reason. The secondary service is only invoked on failure of the primary, and so will not be invoked at all if the primary succeeds. The concurrent combinator executes both services simultaneously, and whichever completes first is returned as the result of the combined service. For both combinators, failure of both service operands results in failure of the combined service. The third combinator for reliability is *repeat.* This combinator repeatedly invokes the parameter service until it succeeds.

Finally, two 'primitive' combinators that require no service operands are *fail* and *stall.* Stall never completes or fails and always has a rate of zero, and fail immediately fails. The complete set of combinators allow the programming of reliable composite services such as concurrent and alternative downloads, delayed repetition, and interpreted failure through rate monitoring and timeout. Consider the following example, presented in [56].

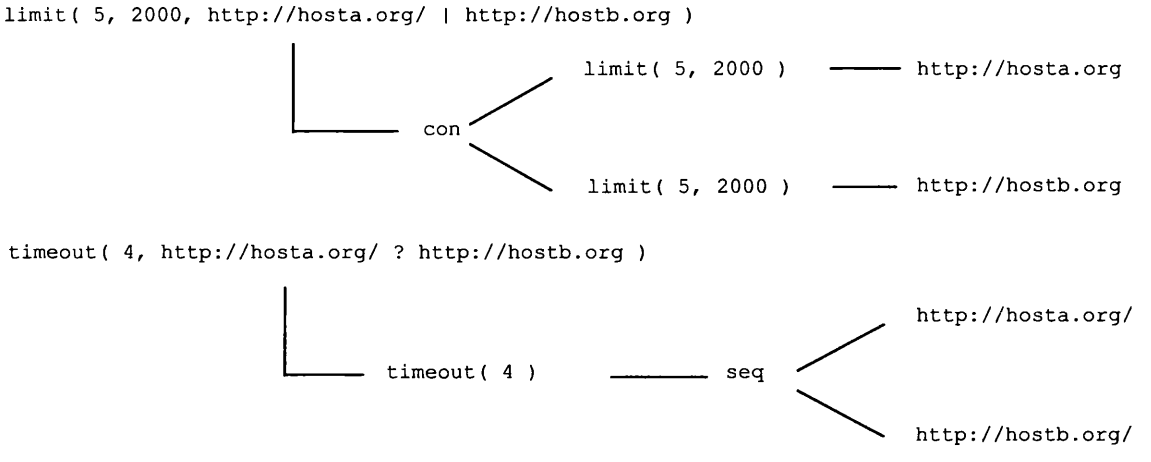
```
let dbc = function(ticker) is
  post("http://www.dbc.com/cgi-bin/htx.exe/squote",
        source="dbcc" TICKER=ticker format="decimals" tables="table")
let grayfire = function(ticker) is
  index("http://www.grayfire.com/cgi-bin/get-price", ticker)
let getquote = function(ticker) repeat(grayfire(ticker) ? dbc(ticker))
getquote("DEC")
```

This program defines two functions for looking up stock quotes based on two different gateways. It then defines a very reliable function that makes repeated attempts in the case of failure, alternating between the gateways. It then uses this function to look up the quote for Digital Equipment Corporation.

The service combinator algebra allows the specification of failure semantics for particular services or combined services with the timeout and limit combinators. The timeout combinator applied to a combined service implies failure of the entire combined service if its evaluation violates the time constraint. This is irrespective of observed times for individual

http services. Conversely, the semantics of limit are such that failure of *any individual basic service* within the limited service (combined or otherwise) is implied if its rate of transfer falls below the specified absolute value. Failure of a service in this way does *not* imply failure of the combined service produced by limit. Thus, limit does not require any notion of the rate of a combined service, but only those individual http services within the combined service.

These semantics can be shown more clearly if we view expressions in the combinator algebra as dynamic trees. The 'evaluation' of an expression amounts to collapsing the tree down to a single service, in a manner defined by the semantics of the non-leaf nodes. The first



parameter of the limit combinator is the minimum rate limit, and the second specifies a startup time for services before rate constraints are applied.

The diagrams show that as we construct the tree, the rate limit propagates down to each basic http service, but that this is not the case for timeout, which constrains combined services as a whole. These semantics for limit side-step the issue of having to define the concept of rate for combined services that are composed of several basic services, some of which may be inactive. The concept of rate can only be sensibly applied to individual basic services – there is no immediately obvious way to limit the rate of a combined service as a whole. The limit propagation design decision has implications for combined service modularity in the context of nested limits. All constraints set by limit are propagated down to the level of basic services, so that in the presence of more than one limit, some services may have more than one constraint applied to them. In this case, the latency and rate constraints are unified according to the respective minimum of the two rate and startup times.

Since limits at a higher level of abstraction override those at a lower level of abstraction if they are more constraining, limits at the higher level can prevent slow downloads from succeeding. This is true even if they are constrained only loosely by their immediate limiter, as in the example above. That is, limits at a high level of abstraction can prevent slow things

from happening at a lower level of abstraction, even if they are deemed to be proceeding at an acceptable rate at that level of abstraction.

One final issue involving rate limits concerns rate variability. In the previous chapter, we saw that even for transfers that are achieving good progress, dynamic rate frequently spikes and troughs. The number of spikes and troughs is inversely proportional to granularity at which dynamic rate is calculated. For our experiments, we chose a granularity window of 700ms. However, there is no discussion of dynamic rate calculation in the service combinator literature, and so we were unable to determine what affect troughs in particular might have on combinator programs. We have implemented our own service combinator algebra, which is true to the semantics defined by Cardelli and Davies. This implementation is available online [58]. Our version of the algebra incorporates some of our own concepts, but these are not relevant here and so are discussed later in Chapter 5 – *A Conceptual Domain for Web Programming*. In writing programs with our service combinator algebra, we found that failure would often be interpreted erroneously, as a result of intermittent troughs in dynamic rate. At the language level, the service combinator algebra does not provide the means to relax rate constraints in order to 'overlook' intermittent rate troughs. As a result we were forced to apply a 'smoothing function' in order to mask brief troughs in rate. We discuss these techniques in more detail in Chapter 5.

WebL - Web Language

WebL (pronounced 'webble'), or Web Language, is a programming language designed specifically for the purpose of automating tasks on the Web. It incorporates a modified service combinator algebra integrated with a general purpose exception handling mechanism. A major feature provided by WebL is a *markup algebra*, designed for computing over the structure of HTML and XML documents. We do not discuss the markup algebra further, since it is designed to address issues arising from the domain of Web content, and we are interested primarily in those language concepts that relate to the Web's failure and performance properties. However, since WebL is one of the few direct attempts at designing a programming language for Web computation, we examine some concepts of the underlying language model additional to those concerned solely with its embedding of service combinators.

WebL is dynamically typed. The only static checking of a WebL program that takes place is with respect to the context free syntax. Thus, the only errors that are detected before execution of a program begins are those of syntax. The one exception to this is errors of undeclared identifier usage, which are detected statically. In WebL, it is neither possible, nor

necessary, to statically denote a type. Instead, there is an association at run-time between every WebL value and a dynamic type.

WebL is block structured with lexical scoping, meaning that identifier bindings[1] are only visible within their smallest enclosing program block. However, values assigned to variables outside of the scope of their creation carry their closure. All WebL values are immutable, with the exception of objects, which have mutable fields. Although all values (with the exception of objects) are immutable, a bound *variable* can be rebound to another value of arbitrary type at any time, so long as it is in scope. Variable declarations do not require the specification of a manifest type, and initialisation is optional. A type is inferred for the variable at the time of its first assignment and until then the value, and type, of the variable is *nil*. Nil is not compatible with any other types, the implicit exception to this being during assignment. WebL is type complete, and all values are first class in that they may be bound to an identifier, form the result of arbitrary expressions, be passed as parameters to a function, and be returned from a function.

Object values are mutable, and are created with an initial set of named fields of any type. Object fields are selected with the traditional 'dot' notation, but the success of field selection depends dynamically on the presence of the named field. No static knowledge is assumed for what fields particular objects may contain, or the type of those fields. New fields can be dynamically added to an object by using a slight variation on the syntax of assignment to an object's indexed field, namely the use of the ":=" operator instead of the usual "=" for assignment.

WebL is not an object-oriented language, since there is no explicit mechanism for creating an object hierarchy by inheritance. However, an *object-style* programming methodology is possible within the WebL object framework by binding functions to the fields of an object. If these functions are declared to be of type *meth* (method), a 'self' identifier is automatically introduced into the scope of the method's body. This identifier is bound to the object from the context of the method's invocation.

WebL incorporates a service combinator algebra based on that defined by Cardelli and Davies. It provides two basic services, *get* and *post* that take as their first parameter a URL in the form of a string, and an object as the second parameter. On service invocation, WebL marshals the object fields into name-value pairs within an encoded query string, then attempts to fetch the document referenced by the given URL with either the http GET or POST method

[1] We shall use the terms 'identifier binding' and variable synonymously in the context of WebL.

as appropriate. As with the service combinator algebra, encoding is equivalent to that of HTML form submission.

The WebL failure model is based on a general purpose exception handling mechanism with exceptions represented by arbitrary WebL objects. During program execution, every operation is checked for correctness, and an exception is raised if this is not the case. The failure model incorporates all forms of type errors and also operational errors such as division by zero and arithmetic overflow. In addition to these system exceptions, WebL provides the facility to define, raise, and handle *user* exceptions.

The raising of an exception causes termination of the active program block and propagation out of static block scope and up the dynamic invocation chain. For any program block that may raise or propagate an exception, the programmer can define a series of boolean guard expressions and associated blocks of code implementing exception handling logic. If an exception propagates out of a guarded block, each guard is evaluated in turn within the context of the exception. The first guard that evaluates to true causes its associated handler to be evaluated and returned as the value of the propagating block with which the guard is associated. Since an exception can be any object, for which there is no static information about its components, guard expressions typically interrogate the structure of an object in order to determine which handler should be invoked. In order to avoid dynamic type errors and the signalling of additional exceptions, the WebL documentation encourages programmers to adopt a uniform convention as to the structure of generated exception objects. However, the language enforces no such convention, thus allowing a degree of flexibility in the construction of variant exception mechanisms.

In addition to the flow control abstraction provided by its general purpose exception handling mechanism, WebL incorporates a modified service combinator algebra. This is integrated with the exception mechanism, and provides the sole means for concurrency in WebL. Interestingly, WebL generalises the combinator concept, allowing arbitrary computations to be passed as service operands in addition to primitive Web fetch services and combined services constituted from Web fetches. The granularity of this is at the block level. The combinators provided by WebL are the same as those defined by Cardelli and Davies: sequential execution, concurrent execution, time-out, repetition, and non-termination. However, WebL does not provide an equivalent to the limit combinator. This elision is based on the fact that arbitrary computations can be passed to combinators, and is motivated by the belief that the concept of transfer rate cannot be applied to arbitrary computations.

The fail combinator is subsumed in WebL by arbitrary exception objects, which are propagated in a similar manner to failure in the service combinator algebra. Failure is

indicated by raising an appropriate exception object. This is a more flexible failure model than with Cardelli and Davies' algebra, since information detailing the nature of failure can be carried with the exception. However, the simple failure representation in the service combinator algebra allows for clean and regular semantics for failure propagation in the presence of service combinators. In contrast, with WebL the presence of service combinators can result in loss of failure information, the extent of which we now go on to detail.

The failure of any WebL service (primitive or arbitrary computation) results in the generation of an exception, the dynamic propagation of which can be affected by the presence of an operating service combinator. For sequential and concurrent combinators in the service combinator algebra, failure of both service operands causes the combined service to propagate failure. However, when both service operands fail with a WebL combinator there are two exceptions, and it is not immediately obvious which of the two exceptions should be propagated. The service operands will in general raise distinct objects, which may even be of differing type. The design decision made in WebL is that for sequential computation, the propagated exception is that arising from the secondary service, and the exception of the primary service is lost. For concurrent computation, the propagated exception is the one arising from the first service operand, and the secondary exception is lost.

The basic services defined by Cardelli and Davies are non-deterministic but atomic. They return a complete result, do not terminate, or explicitly fail. Since the service combinator algebra is intended to be used either alone, integrated with a functional language, or modularly embedded within a general purpose language, there is no side effect caused by failed services. However, WebL allows arbitrary computations to be passed to combinators as service operands, and because WebL is imperative these may cause side effect. WebL does not undo any side effect caused by services that fail part way through their execution. This has implications for the sequential combinator.

In WebL, if the primary service of a sequential combinator computation fails, the computation performed by that service before the point of failure is not discarded (with the exception of update to variables declared in the local block scope of that service). There is no automated roll back or facility for cleanup, and control passes directly to the secondary service. If the primary updates free variables, and it must if results of its computation are to be retained beyond its activation, those updates are visible to the secondary service, even if the primary fails. This means that the behaviour of the secondary service can be dependent upon computation performed by the primary service, since update by the primary to variables common to both of their scopes is exposed in the secondary. Despite this, the nature of the failure in the primary cannot be determined directly by the secondary, since the exception raised by the primary is not available to the secondary. On success or failure of the secondary

service, the programmer is responsible for cleaning up the computation of the failed primary service. This could be done in either the secondary service itself since failure of the primary is implicit at that point, or in the scope that invoked the combined service. Resource cleanup is simplified by the fact that WebL has automatic memory management (garbage collection).

The sequential combinator has behaviour analogous to a restricted form of termination model exception handler. That is, the secondary service is the 'handler' for an exception, but the details of that exception is not available. In the program fragments below, we show that semantics equivalent to that of the sequential combinator can be implemented with the WebL exception mechanism alone.

```
let seq = fun(primary:fun() → void; secondary:fun() → void) {
  try { primary() }
  catch(e) { secondary() }
}
```

In this context, the following program:

```
let pri = fun() { ...p }
let sec = fun() { ...s }
seq(pri, sec)
```

is equivalent to, but syntactically more verbose than:

```
{ ...p } ? { ...s }
```

The WebL concurrent service combinator is the only means for expressing concurrency in WebL. The WebL concurrent combinator differs from that defined by Cardelli and Davies, in that it does not have strict alternate semantics. With Cardelli and Davies' algebra, the concurrent combinator returns the result of whichever service completes first, and the computation of the other service is pre-empted and discarded. WebL is similar in that the result of the concurrent combinator is the value (if any) of whichever concurrent block finishes execution first. However, both blocks must complete (or one or both fail) before execution continues at the statement following the concurrent combinator. Another difference

is that in WebL, concurrent computations are not automatically transactional or mutually exclusive, and can potentially be cooperative and blocking. However, a variable locking mechanism is provided that allows the specification of critical sections for concurrent cooperating computation.

Summary and analysis

Service Combinators integrate the Web fetch primitives with combinators that allow constraints to be set on rate, latency, and time observables of Web fetches, and specify possibly concurrent flow control for failure should any of the constraints be violated. The flow control mechanism is a simple concurrent extension of an exception handler that is limited by the fact that no failure information is carried by the failure representation. Combined service expressions are modular in that the pattern of flow control within them is hidden from the abstraction level that invokes them. This means that a programming language that embeds the service combinator algebra cannot integrate the service combinator flow control and concurrency mechanisms with its own mechanisms. The Service Combinator algebra successfully integrates the concepts of domain exposure, failure interpretation, flow control for failure, and concurrency. However, the resulting programming model is 'closed' in that it cannot be embedded seamlessly in another programming language, even one that is similar to the combinator algebra itself.

WebL attempts to embed Service Combinator concepts in a general purpose imperative programming language with a traditional exception handling mechanism, in order to add flow control appropriate to Web computation. However, the exception mechanism interacts with the 'computation constructors' that are the service combinators embedding in a way that can hamper recovery from failure. Primarily this is due to the fact that there is no automatic rollback of computation performed by failed or aborted services, and that manual undoing of computation is hampered by loss of exception information. Exception masking is complicated by the fact that the system may be left in an arbitrary state by an arbitrary number of concurrent threads that may or may not have failed at an arbitrary point.

In this thesis, we are particularly interested in how the means to express what constitutes failure relates to the exposed properties of the Web domain, and in how flow control after failure integrates with concurrency and failure detection. For the class of applications we are interested in, we have identified the following design goals for a domain specific language:

- *Exposing the properties of the Web domain* – any exposure of domain concepts should be orthogonal in that it composes in sensible ways with the rest of the language.

- *What constitutes failure* – the mechanism for failure interpretation should be flexible and orthogonal.
- *Flow control after failure* – the flow control mechanism should integrate with the rest of the language and in particular its concurrency mechanism.

In the Service Combinator algebra, all major observables of the domain (time, rate, and latency) are exposed only within the algebra, and not outside it. Programmers cannot reason about the domain outside the context of a combinator expression. Thus, although it could be argued that this maintains orthogonality, it is an austere orthogonality since the domain exposure does not compose directly with the language in which service combinators are embedded.

The service combinator algebra has the observables of time, rate, and (indirectly) latency. Failure interpretation is achieved by setting constraints on the valid quantities that these may take at run time. What constitutes failure for a particular Web fetch is specified by all the limit, timeout, and retry combinators on the path from that computation to the root of the combinator expression. However, failure can be interpreted only in a limited manner, based on a simple relationship. For example, for any single fetch the most sophisticated constraint implies failure on violation of any one of a maximum latency *or* a minimum rate *or* a maximum time. No relationship other than 'or' can be specified between constraints. The failure representation is a single failure event value, and so carries with it no information as to the nature of the failure.

The failure model for the service combinator algebra is distinct from that of the language in which it is embedded. It is not possible to integrate them, since all flow control within a combined service is hidden. After evaluation, all service combinator expressions represent either a single resource, or failure. The pattern of flow control that gives rise to a particular result depends on the nesting of combinators, and cannot be determined outside the combinator expression. Flow control, concurrency, and failure interpretation are intrinsically integrated. The failure model is independently understandable, but incapable of composing with the flow control and concurrency of a host language. Also, there is a question as to whether the service combinator algebra's concurrent flow control mechanism ever could integrate with the host language, since there is no obvious way to apply rate limits to arbitrary computation.

WebL does not expose the properties of the domain with respect to failure and performance. The only 'observable' present is that of time, which is no more than any general purpose programming language. Transfer rate and connection latency observables are elided from the

failure model due to a belief that they are concepts that cannot be generalised to all computation. In a sense, the domain specific concepts are orthogonal in that they do compose in a sensible way with the rest of the language. However, there is only one domain concept in WebL. Time, which is not particularly domain specific.

In WebL, what constitutes failure is programmable with arbitrary program logic and indicated by raising of an exception, as it is with many general purpose programming languages. However, WebL incorporates a class of automatically generated system exceptions that correspond to manifest failures of Web transfers. The presence of a timeout combinator allows failure to be interpreted for an arbitrary computation (that may or may not consist of Web fetches) if it violates a time constraint. Failure representation is flexible, in that exceptions are arbitrary objects, and the mechanisms for failure interpretation (raising an exception and timeout) are orthogonal. However, failure interpretation is inflexible due to the limited domain knowledge that is available to make failure interpretation decisions.

The WebL flow control mechanism is an attempt to integrate a traditional exception handler mechanism with Service Combinators. Although the exception mechanism is flexible, it does not adhere to the semantics intended for Service Combinators by Cardelli and Davies with respect to atomicity. Thus, the flow control mechanism as it pertains to Web programming does not integrate cleanly with the rest of the language, or with the concurrency mechanism that is provided solely by the concurrent combinator.

Gaining insight from the design decisions in Service Combinators and WebL, in the next chapter we attempt to 'plug the gap' with respect to failure interpretation in general purpose exception handling mechanisms. We attempt this without recourse to domain-specific language level constructs, using only common general purpose programming language concept.

4: Web Fetching with GP Languages

Both Service Combinators and WebL are domain specific languages for programming Web applications, but the related literature specifies no *raison d'être*. That is, the literature does not justify the requirement for a domain specific language motivated by inappropriateness of general purpose languages. Thus, before we develop yet another specialised Web language, it is worth considering whether we can achieve the criteria described at the end of the previous chapter within the context of traditional programming models.

The Libwww API [59], which has several language bindings, allows programmers to set a global latency timeout, which applies to all further download attempts. Individual fetches are not parameterisable. The Java standard Web package provides no direct support for latency or transfer timeout. The Web package is probably implemented using the Java socket facilities, and a global connection timeout can be set for all sockets. However, there are no guarantees that the Web package is implemented with Java sockets, since in the future they may be optimised into native code. However, connection (latency) timeout can be explicitly implemented using concurrency. Transfer timeout can be implemented easily since the Web fetch abstraction is stream based and bytes must be explicitly read from the stream. Programmers can check system time and calculate rate between buffer reads. Neither Libwww nor Java explicitly provides the kind of Web programming abstraction that we desire.

In this chapter, we attempt to 'plug the Web programming gap' in GP programming languages by addressing the perceived weaknesses of domain exposure and flexible failure interpretation. This assumes that flow control for failure is given in the form of an exception handling mechanism. We do not develop new programming language constructs, but instead develop a methodology for Web programming with GP languages that mostly concentrates on domain exposure and on providing flexible failure semantics (FFS).

Failure Issues for a Simple Web Fetch Abstraction

In this chapter we will present several example programs. Throughout, we assume a simple Algol-like language, with first-class functions and a simple termination model exception mechanism, where exceptions are simple names, and are automatically propagated through the dynamic invocation chain until they are handled[1]. The implementation of a simple Web fetch procedural abstraction for text and HTML is shown below. We do not claim that this is an ideal implementation, only that it is appropriate and our best attempt.

[1] The properties of exception handling mechanisms are discussed in detail in Chapter 8.

```
let webFetch = function(URL url → string)
{
  try {
        let ip = lookupDNS(url.host)
              let soc = openSocket(ip, httpPort)
  } catch(...) throw connectionError    //catch all exceptions

  let result = ""                    //accumulator for streamed data
  try {
        soc.write("HTTP 1.1 GET " + url.path)
        while soc.isOpen() or not soc.isEmpty() do
              result := result ++ soc.read(1000)     //read 1000 bytes max
  } catch(...) throw socketError          //catch all
  return result
}
...
let doc = webFetch(new URL("http://foo.org/"))
```

Within webFetch, there are four points at which absolute failure can occur:

- Failure to resolve the hostname to an IP address.
- Failure to open the socket due to manifest network failure.
- Failure in writing the http request to the socket stream.
- Failure during a socket stream read.

These absolute failures are all detected internally to the provided DNS and socket abstractions, and the webFetch abstraction recasts them in a form appropriate for handling by its own invoker. That is, it abstracts over the nature of absolute failure by classifying the four possible kinds of failure into two: socket failure and connection failure. Thus, the failure semantics for webFetch is that failure is constituted by socket or connection failure, and the action taken is the raising of the corresponding exception. These failure semantics capture all forms of absolute failure at the network level that can occur when fetching a Web document, but there are classes of possible failure for webFetch that are not absolute. For example:

- No response to connection attempt (server failure or not present).
- No stream response to GET command (server sends no bytes).
- Cessation of byte streaming at an arbitrary point (server or network failure).

In principle, these failures are undetectable due to the fact that they cannot be distinguished from long delay. Under these circumstances, failure must be *interpreted*, by timeout, for example. There is another class of failure based on the notion of *acceptability*. For example, a server may fail to respond for a period of time that is deemed unacceptable. After the acceptable period has expired, failure to achieve acceptable *performance* occurs, even if ultimately the server does respond. To summarise, we classify failures into three categories:

- *Absolute* – manifest failures in the network or high-level server errors such as document not found.
- *Interpreted* – lack of response at some point, for which it is impossible in principle to determine whether or not absolute failure has occurred.
- *Performance* – unacceptable performance for some aspect of the fetch. Performance aspects directly relate to the Web transfer *observables* we identified in the introduction. These are connection latency, transfer time, and transfer rate.

All absolute failures are captured by the web fetch abstraction, all undetectable failures must be interpreted, and all performance failures result from the violation of some constraint on observable properties of the fetch. The process of interpreting undetectable failures is the same as the process of detecting performance failure. That is, we interpret undetectable failure on the violation of constraints on observables. For example, if a particular Web fetch has failed to connect after sixty seconds, this implies that the server or intermediate network has failed. A latency of sixty seconds would almost certainly also violate performance constraints. Since undetectable failures manifest themselves in the form of 'poor' performance, we can coalesce interpreted and performance failures under the single banner of *failure by constraint violation*.

Before continuing, we should note that socket implementations provided by operating systems generally map 'undetectable' failures onto absolute failures by imposing an

underlying timeout. However, these timeouts are of the order of minutes, and in general we are interested in interpreting failure long before they expire.

Implementing Failure Semantics

Since a Web fetch abstraction does not know the context of the fetch it is making, it cannot be expected to interpret failure and take appropriate action on behalf of the invoker. There is an exception for absolute failures, for example when a document or host cannot be found, since sensible default behaviour can be exercised, such as raising an exception as in the example above. However, when failure is not absolute, for example when a Web server fails to respond within a given period or when document transfer rate drops unacceptably, failure must instead be interpreted. We wish to specify this failure interpretation in a general way. Thus, the abstraction must somehow be parameterised on each invocation with information as to what constitutes failure, and what action is to be taken when it occurs. Such parameterisation specifies the particular failure semantics for that particular Web fetch.

In the previous section, we concluded that failure interpretation should occur on the violation of dynamic constraints set on the observables of each Web fetch. Service Combinators and WebL (the latter only to a limited degree) expose the properties of the domain explicitly at the language level with *observables*. However, since we are implementing our Web fetch abstraction with a general purpose programming language, we must implement a mechanism to expose the properties of the domain ourselves, with program logic internal to the Web fetch abstraction. It is this program logic that implements the 'how' for failure interpretation, which is parameterised with the 'when' of constraint values. At this point we must decide on the different forms of constraint violation we require. Earlier, we identified the various performance aspects of Web fetches as latency, transfer time, and transfer rate. We constrain these according to the following.

- Maximum latency time in seconds.
- Maximum transfer time in seconds.
- Minimum rate in bytes per second.

The units of measurement (seconds, bytes per second) here are somewhat arbitrary, but not particularly relevant to the mechanism, apart from the fact that constraint values must be specified in terms of the same measurement unit. Whatever the measurement, it can be expressed in terms of floating point numbers. Logic internal to the abstraction checks these

constraint values against a dynamically calculated rate, time and latency, and automatically generates an appropriate exception on their violation.

This takes care of failure interpretation. To complete the specification of failure semantics, we statically associate an exception handler with the abstraction. This directs flow control after the detection of failure. The Web fetch abstraction with parameterised constraints on observables follows.

```
let webFetch =
function(URL url, float minRate, maxTime, maxLatency → string)
{
  try {
        let ip = lookupDNS(url.host)
        let soc = openSocketTimed(ip, httpPort, maxLatency)
  } catch(...) throw connectionFailure

  let result = ""
  try {
        let startTime = time()
        soc.write("HTTP 1.1 GET " + url.path)
        while soc.isOpen() or not soc.isEmpty() do {
              let t = time()
              result := result ++ soc.read(1000)     //read 1000 bytes max
              let dt = time() - t
              let rate = soc.numBytesRead() / dt
              if t - startTime > maxTime do throw timeoutException
              if rate < minRate do throw rateException
  } catch(...) throw socketFailure
  return result
}
...
let url = new URL("http://foo.org/")
let doc = webFetch(url, 10.0, 25.0, 5.0)
catch(connectionFailure) { ... }          //flow control after failure
catch(socketFailure) { ... }
```

In the main loop, programming logic repeatedly calculates the current rate and elapsed time of the Web transfer. These are then compared against the constraint values and an exception thrown if current time is greater than the time constraint or if current rate is less than the rate constraint. In Chapter 2, we saw that dynamic rate is prone to fluctuation. Smoothing techniques can be applied to diminish the impact of intermittent rate troughs for which failure should not be interpreted. For brevity we do not show implement this here. In Chapter 5, we discuss smoothing techniques in more detail.

The parameter constraints on the dynamic values for observables form a simple 'or' relation for interpreting failure. That is, failure is implied on violation of either the time *or* rate *or* latency constraints. This may be sufficient for many purposes, but if we wish to imply failure by a more sophisticated constraint relationship, it is not immediately obvious how we can supply this information to the Web fetch abstraction. For example, to change failure interpretation to an 'and' relationship between constraints, say, there are two obvious alternatives. First, we can modify the internal logic of the abstraction to reflect the new constraint relationship, and by copy and paste implement another fetch abstraction with the new relationship.

```
webFetch : function(URL url, float minRate, maxTime, maxLatency → string)
webFetchAnd : function(URL url, float minRate, maxTime, maxLatency → string)
...
let doc = webFetchAnd(new URL("http://foo.org/"), 10.0, 25.0)
```

This method duplicates code, and if the relationship is complex, it may be difficult to describe in the abstraction's name, forcing reliance on auxiliary documentation. Moreover, it assumes that the code for the original abstraction is available. The second alternative is to parameterise a single abstraction by the appropriate constraint relationship. We have:

```
type constraintRelation is enumeration [ orRelation, andRelation ]
webFetch : function( URL url, float minRate, maxTime, maxLatency,
                     constraintRelation c → string)
```

The following invocation parameterised a Web fetch with different constraint values, and specifies that constraint violation occurs on simultaneous violation of all constraints.

```
let doc = webFetch(new URL("http://foo.org/"), 10.0, 25.0, 5.0, andRelation)
```

The Web fetch abstraction can be provided with any number of associated constraint relationships defined. However, as the number of constrainable observables increases, the number of possible constraint relationships becomes extremely large. Assuming that the constrainable quantities (rate, latency, and time) are all of a single type, in this case they are floating point numbers, the number of possible constraining expressions is lower bounded by n^{bi}. Here, n is the number of quantities, b is the number of infix boolean operators in the host language, and i is the number of inequality operators. Although many of these permutations are relatively unlikely, such as a constraining expression involving a maximum rate, for example, and many pairs of relationships are equivalent in meaning, it is unreasonable to restrict the programmer to only those constraint relationships that the abstraction designer deems useful. Instead, it is desirable to allow maximum generality, with the specification of an arbitrary constraining expression for each Web fetch.

An Approach with Higher Order Functions

We present a methodology that allows the programmer to parameterise a Web fetch abstraction with a constraining expression directly. Our methodology involves the use of higher order functions (or first-class functions, implicitly). We require the following declarations.

```
type Constraint is function(float rate, time, latency → bool)
webFetch : function(URL, Constraint → string)
```

webFetch repeatedly evaluates its constraint parameter while the document downloads, passing to it the dynamically calculated latency, rate, and elapsed time values. It raises an exception if the constraint function ever returns false. The following code implements a Web fetch for which the conditions of failure are met on the truth of an arbitrary expression (in *italics*).

```
let myConstraint = function(float rate, time, latency → bool) is
        return not (latency > 5.0 or (time > 20.0 and rate < 10.0))

let doc = webFetch(new URL("http://foo.org/"), myConstraint)
```

It does this without modifying the Web fetch operation, which consequently can be provided in a library, without source. In our example, the constraint values are hard coded into the expression. For generalisation, the manifest values in the constraint expression should be parameterisable. We achieve this with a constraint function generator. In this way, any number of specialised constraints can be generated from a single template.

```
let myConstraintGen =
    function(float minRate, maxTime, maxLatency → Constraint) is
        function(float rate, time, latency → bool) is
            return not (latency > maxLatency or
                (rate < minRate and time > maxTime))

let myConstraint = myConstraintGen(5.0, 20.0, 10.0)
webFetch(new URL("http://foo.org/"), myConstraint)
```

On violation of the constraint, webFetch raises a socketError exception. However, this carries little information as to the nature of failure. For example, on failure, the invoker of a download cannot determine whether failure was absolute or interpreted by the constraint, and if the latter, how the constraint was violated. We can implement a mechanism that allows such determination with our exception handling mechanism. Here, the constraint function returns nothing, but may raise an exception that webFetch will propagate to its invoker.

```
let myConstraint = function(float rate, time, latency → void) is
    if latency > 5.0 then throw latencyException
    else if time > 20.0 and rate < 10.0 then throw timeRateException

try webFetch(new URL("http://foo.org/"), myConstraint)
catch latencyException { ... }
catch timeRateException { ... }
catch { ... }                    //catch absolute failures raised by webFetch
```

In the example, we have embedded the constraint values within the constraint function. However, we can apply constraint generators as before, in order to allow parameterisation by constraint values. The incorporation of exception handling completes the means for full parameterisation of failure semantics.

Since we now have a means to express arbitrary relationships between constraint, we can introduce new observables that alone are not directly performance related, but may be useful for interpreting failure in conjunction with other observable. Such an observable is *download completion percentage.* As soon as the header information for any http resource is downloaded, the size of the resource in bytes is known. The amount of a resource downloaded at a particular time during transfer may influence the decision as to whether or not failure should be interpreted. For example, a programmer may specify constraints on time and rate for a particular transfer, but be willing to relax these constraints should the download be close to completion. We define the completion observable to be a number between zero and one that reflects the proportion of resource that has been downloaded. Given the definition:

```
type Constraint is function(float rate, time, latency, completion → bool)
```

We can create constraints of the form:

```
let myConstraint = function(float rate, time, latency → bool) is
    return not ((time > 20.0 or rate < 10.0) and completion < 0.75)
```

This enforces a constraint on time and rate only if more than 25% of the document has yet to be downloaded.

Use of Methodology with Object-Oriented Languages

Many modern languages used for programming Web applications, notably Java, do not provide higher-order functions. This precludes direct use of the technique described above. However, object-oriented languages allow an approximation of the methodology using dynamic binding. Instead of a function, a constraint is an object with a single method that causes evaluation of the constraint. Consider the following code, in Java-like syntax.

```
class WebServices {                          //in library
        ...
        public static Document webFetch(URL url, Constraint c)
                raises ConstraintException, WebException { ... }
        ...
}

class ConstraintException { ... }    //in library

class Constraint {                           //in library
        abstract void eval(float rate, float time, float latency)
          raises ConstraintException;
}

class MyLatencyException public ConstraintException { ... }

class MyConstraint {
        private float maxLatency, minRate, maxTime;
        public MyConstraint(float _maxTime, float _minRate, float _maxLatency)
      {...}
        public void eval(float rate, float time, float latency) {
                if(latency > maxLatency) throw new MyLatencyException(...);
                if(rate < minRate && time > maxTime)
          throw new ConstraintException (...);
        }
}

...
WebServices.webFetch(     new URL("http://foo.org/"),
          new MyConstraint(5.0, 10.0, 20.0));
```

There is more syntactic overhead in the creation of constraints than when using higher-order functions, but importantly, the *use* of constraints has similar minimal overhead.

Analysis

Our methodology allows the specification of constraints on the values of dynamic Web fetch observables for which failure is implied should they be violated. For a Web fetch abstraction, the constraint values, the boolean logic of each constraint, and the relationship between constraints are all parameterisable. Our methodology provides the means for expressing what constitutes failure in a manner that is more flexible than that of the Service Combinator algebra and WebL. The FFS model allows Web fetch abstractions to be parameterised with the 'how' for failure interpretation, as well as the 'when'. Moreover, since our methodology allows for exception generation to be associated with the constraint logic rather than the Web fetch abstraction, flow control for failure is also parameterisable. The nature of a particular instance of failure is then available outside the Web fetch abstraction. In contrast, Service Combinators do not and cannot distinguish between different forms of failure, since the failure representation is minimal and the flow of control after failure hidden.

As they pertain to our FFS methodology, the three criteria for Web programming systems introduced in the last chapter are:

- *Exposing the properties of the Web domain* – The FFS domain concepts are embedded within the Web fetch abstraction, so the domain is not exposed explicitly. To an extent, it is possible to reason about the domain in general programming logic, but this reasoning is limited to the construction of the constraints, and analysis of information returned in exceptions generated by the Web fetch. The level of domain exposure is limited because we cannot compute with, express, or store actual domain quantities. However, the domain exposure that there is is orthogonal since constraints are quantified with floating point numbers and constraint relationships are expressed in the boolean algebra of the programming language.
- *What constitutes failure* – Failure is interpreted by specification of an arbitrary boolean algebraic expression involving constraints on observables, and. is constituted by violation of these constraints. Failure is represented by exceptions in accordance with mechanism of the language in which methodology is applied. Failure interpretation is extremely flexible, and this is certainly the strongest aspect of our methodology.
- *Flow control after failure* – Flow control after failure is achieved by exception propagation out of the Web fetch abstraction. The particular exceptions that are generated for different failure conditions is a parameter of the Web fetch abstraction. However, this

is not the case for programming languages that require static exception interfaces[1], since the Web fetch abstraction must declare the number and type of exceptions that it may generate. This means that the designer of the Web fetch abstraction chooses the particular exceptions that can be generated, but the programmer can still influence flow of control after failure to an extent, since they specify the logic of the exception handlers. In any case, the mechanism for flow control after Web failure integrates with the flow control for failure mechanism for rest of the language, since they are both the same exception handling mechanism.

There is a significant drawback of our methodology with regards to integrating concurrency and flow control for failure. Primarily, this is a result of the fact that an exception handling mechanism is essentially a serialised model of flow control that does not directly extend to a concurrent context. WebL is an example of what happens when attempting to integrate exception handling and concurrency, and exception propagation in the Service Combinator algebra only works because it has the simplest possible failure representation.

There have been other attempts at integrating exception handling and concurrency. In the Oz programming language [60], exception handling is thread-wise. Concurrent threads escape their enclosing try-scope. The following code will execute computations S1 and S2 concurrently:

```
thread S1 end S2
```

However,

```
try thread S1 end S2 end
```

is equivalent to

```
thread S1 end
try S2 end
```

[1] See Chapter 8 – Exception Handling.

That is, the block S1 escapes the exception context, the exception block being recursively propagated down until it reaches a non-concurrent context. This amounts to the fact that exceptions can not be defined in a concurrent context.

Arche [7], Ada 95, Ada 83, Modula-3 [15], SR [6], Real-Time Euclid [12], and Java are all languages with concurrency and exception handling mechanisms. However, in all of them, the exception handling mechanism is serialised, defined only on a thread basis. Some languages allow programming of concurrent exception resolution since the handler can communicate with other computations, but this is complex. Other languages (such as CSP and Ada 83, for example) simply have no features to interrupt or in any other way asynchronously inform the participating computations when one of them has raised an exception.

Recent research by Romanovsky [61] outlines the state of the art in concurrent exception resolution, and proposes methodology that is implemetable with the Java exception handling mechanism. Although the methodology allows sophisticated resolution logic, it is limited by the fact that all exceptions must be of a standard form and cannot be parameterised. The methodology is based on the concept of concurrent atomic actions, which we return to in Chapter 7 – *Analysis of Related Work.* Concurrent atomic actions provide a heavyweight solution to the problem of automated error recovery and are primarily intended for distributed computing. Since we do not wish to address the issue of distribution directly, we feel that they are inappropriate for our needs, which are for lightweight threading only.

In conclusion, we have presented a methodology for implementing flexible failure semantics in GP languages based on higher order functions. There is no evidence of any other failure interpretation techniques in use that are as flexible as the FFS model. However, the FFS model fails to fully integrate failure interpretation, flow control for failure, and concurrency, because exception handling is a serialised model. The fact that there is no programming language that integrates exception handling and concurrency to the extent we desire suggests that we should take an approach that is not based on GP exception handling mechanisms. Since exception handling is in mismatch with our intended programming domain, we intend to develop a language model from the ground up, with the goal of integrating failure interpretation, flow control for failure, and concurrency.

5: A Conceptual Domain for Web Programming

In the first chapter, we stated that in asynchronous distributed systems failure detectors utilising timeout alone can only be approximations. This is because there are classes of system and network failure that cannot be distinguished from slowness. Consequently, timeout is not an accurate means of failure interpretation. In addition to inaccuracy, the use of timeout as the sole means of failure interpretation can also lead to inefficiency. This is due to the forced formulation of an upper bound of execution time for the timed computation. Programmers must estimate a value for timeout that accounts for the worst possible case – the maximum possible time passing during execution of the service before which failure can be interpreted unambiguously (in pragmatic terms). This means that the required timeout must be significantly longer than the *mean time until failure*, which is the average of the times at which failure is actually detectable. This can cause substantial delay in execution, while waiting for the timeout of a failed computation to expire before taking remedial action. For example, we can specify timeouts of ten minutes for all fetches. Programs would still work, and failure interpretation would likely be very accurate in that failure would rarely be incorrectly assumed. However, this is grossly inefficient, and exacerbated by the fact that failure is a common occurrence on the Web.

In developing a programming system for Web computation, we intend to provide accuracy and efficiency when *interpreting* the failure of non-deterministic computations. Improving the accuracy of failure interpretation alone leads to increased efficiency, since the sooner one can be certain of failure, the less time is wasted before taking remedial action. Our approach identifies a set of observables additional to timeout that can be used together in determining the *likely future behaviour of Web transfers* – a notion that is important for accurate failure interpretation. In making a determination of future behaviour, we see it as fundamental that the more information available, the more accurate any failure interpretation.

In a Web programming system, perhaps more important than accuracy and efficiency is *flexibility* in failure interpretation and flow control. In the Web domain, the concept of failure itself is paradoxically unimportant. That is, failure need not even be defined if there is sufficient means for the programmer to express classes of program behaviour based on interpretation of observables. In this domain, 'failure' is failure to produce a document within certain parameters, which can only be specified by the programmer. Exception handling mechanisms in traditional programming languages make available certain classes of behaviour, and these are intended for 'exceptional' or 'failure' circumstances that are absolute. However, in the Web domain failure is not exceptional in any way, and it is rarely absolute. With observables, we gain the ability to specify flexible patterns of interpreting

behaviour, in addition to 'better' detection of 'failure' with less time wasted in making the detection.

Before attempting to design novel programming language abstractions for Web programming, we must consider exactly what it is we wish to compute over. As we saw in Chapter 2 – *Analysing Web Failure and Performance*, the global network is inherently unreliable in that connections can arbitrarily fail to initiate, and data transfers can fail at any time and are subject to largely unpredictable fluctuations in bandwidth and latency. We require failure interpretation based on observables so that we can distinguish between slow and failed Web queries more accurately than with timeout alone. Traditional programming languages are not designed primarily for the task of computing over the Web. The multiplicity of failure modes, and the difficulty in detecting some of them motivates the development of new concepts that relate to the Web's computational properties. These concepts can then be incorporated into programming languages designed specifically for Web computation.

The notion of 'what we can compute over' is referred to as a *conceptual domain.* A conceptual domain is a set of concepts such as integers, structures, and arrays, for example, in terms of which programming languages are defined. Programming languages draw from one or more conceptual domains in order to define their *universe of discourse*, which is the set of concepts that can be legally expressed in the programming language. The universe of discourse is always a subset of the union of available conceptual domains. For example, three-dimensional arrays might be present in the conceptual domain, but cannot be expressed in the language so are not in the universe of discourse. Programming languages are concrete *computational models* describing denotations that allow us to express computations over the universe of discourse. Since conceptual domains are the foundations on which we construct languages, the mapping from the computational model into the conceptual domain becomes more difficult the further removed the domain from the type of computations we wish to perform. That is, the design of a programming language is aided by a conceptual domain that is appropriate for the intended application class of that programming language. In this chapter, we describe a conceptual domain designed specifically as a foundation for programming languages that map high-level Web programming abstractions to the domain.

According to Cardelli, *observables* are "the events or states that can in principle be detected" [62]. This definition is borrowed from its use in 20th century physics where it was formalised in the Copenhagen Interpretation of quantum physics. In classical physics, the observables of a system such as particle energy and momentum, for example, are considered well defined entities that change their values over time according to certain dynamic laws and which can in principle be observed without disturbing the system itself and thus distorting their quantities. It is a fundamental finding of quantum physics that this is not the case. Although the same is

certainly true for the web observables we are interested in, we do not believe that the act of measuring them will alter their values significantly enough to compromise any results we obtain.

We aim to extend the concept of observable so that it can form the basis of an algebra for reasoning about the dynamic behaviour of computations involving the Web. That is, we wish to define a conceptual domain for Web programming languages based on the concept of observable. For now, we identify the particular observable properties of individual http connections that might be useful in interpreting failure to be *transfer rate*, *elapsed time*, and *latency time*.

In the Service Combinator algebra, WebL, and our FFS model, programs that interpret failure in some way contain manifest values corresponding to acceptability limits on some or all of these observables. For example, a program might contain constraint values corresponding to a maximum time and minimum rate. In WebL and Service Combinators, there is an 'or' relationship between these constraints, meaning that failure is interpreted on violation of either constraint. However, the FFS model examined the possibility of using other relationships, such as an 'and' relationship, for example, where failure is interpreted only on the conjunction of two or more constraint violations. A major design goal of our conceptual domain is to allow flexible failure interpretation similar to that of the FFS model by allowing constraints on a number of observables with potentially sophisticated relationships between those constraints. Our conceptual domain is to achieve this, but in a manner that is more amenable to integration with concurrency mechanisms.

Persistent Relative Observables

The basis of our conceptual domain is the *persistent relative observable*, which is an extension of the basic observable. Persistent relative observables are quantities, observable with respect to some computation, whose values are determined relative to a historical context for previous executions of the same computation. Since the properties of the particular observables we have identified are directly related to individual http transfers, we are interested in the non-deterministic computations that fetch Web documents. However, because the result of an observation is calculated relatively, we can generalise the concept to all computation, and not just fetches. That is, for rate defined in bytes per-second, say, we can assume that all deterministic computations execute at rate x. This means that observations relative to a historical context exhibit the same relative value for rate (one) every time they are executed, side-stepping the issue of having to define the notion of bytes per-second for arbitrary computation.

When accessing the Web, it is difficult to infer any meaning for the quantities of observables if there is no historical context to that particular access. For example, given a value for the rate of a Web transfer, qualitative assessments such as 'too slow' or 'very fast' are not particularly meaningful unless there is a context of comparison with previous invocations of the same or *equivalent* Web transfers. This is due to differences in bandwidth and geography across the Internet. 'Equivalent' here could mean a fetch from the same physical server in the case of rate and latency, but must refer to a fetch of the same physical document in the case of elapsed time since download time is related to document size.

When writing a program with Service Combinators, WebL, or the FFS model with the intention of addressing robustness, the programmer quantifies what constitutes failure in terms of constraints on observables. These constraints are specified in terms of absolute units of measurement, such as bytes per-second or seconds. For example, failure for a particular download could be implied if the transfer rate falls below five kilobytes per second, and the value five is manifest somewhere in the program source. What is critical here is that this value makes a concrete assumption about the speed of the local network connection. Failure assumptions are based on network context: five kilobytes per second may be a slow transfer rate for a download to a system with T1 connectivity. However, this transfer rate is fast for a system having only a modem. In general, failure semantics specified for a computation on one machine may result in entirely different behaviour on another, because the notion of 'acceptable performance' is entirely different. Programs specifying constraints on observables in terms of absolute units of measurement cannot be reliably executed on different machines without source-level modification, or without complex programming logic that explicitly takes account of the network context. By the same argument, the use of absolute measurement units poses problems for programs that are mobile in the network. Finally, absolute measurement units compromise program longevity, since over time the bandwidth of the global network is increasing and latency decreasing, and with these changes the notion of what is acceptable performance. For programs to be portable, mobile, and future proof, computation with observables should usually be independent of absolute measurements.

The values of persistent relative observables associated with a Web fetch are calculated relative to the average[1] value observable on previous executions of equivalent Web fetches. That is, the act of querying an ongoing Web fetch at any point in its duration results in a floating point number that is a ratio of the current observable value relative to the average value exhibited on previous invocations. For example, if the relative rate observable of a Web fetch is seen to be 0.5, this indicates that the fetch is proceeding at half the rate it did, on

[1] We leave unspecified whether this average is the mean, median, or even some other function.

average, during previous transfers. Similarly, if the relative time of a Web fetch is seen to be 2.3, say, this indicates that the fetch has so far taken approximately twice as long as previous attempts, and has still not completed. Persistent relative observables provide a historical context to observations that is independent of absolute measurement units.

An important concept in interpreting failure is *acceptability based on the norm*. That is, failure is only interpreted in circumstances where the observable properties of a Web fetch deviate from those that are to be expected. Our persistence mechanism maintains a historical context for all Web fetches, and when observations of ongoing Web fetches are made, it compares the values of current dynamic observables with those in the historical context. We define 'normal' or 'acceptable' transfer progress to be when a transfer's current observables are similar to those in the historical context. Querying our mechanism with respect to a particular transfer returns a value that is the ratio of the current absolute observable value to the historical context for that transfer. There are exceptions to this for some observables, but in general, a result close to 1.0 indicates acceptable progress, by our definition.

In order to remove any ambiguity before presenting examples, we define each observable in some detail:

- We define persistent relative **elapsed time** as the ratio of the current elapsed transfer time to a single value that represents the historical context for elapsed time. The historical context is calculated by the average of all download times for that URL. The current and historical times must be expressed in terms of the same measurement unit.

- We define persistent relative **latency time** as the ratio of the current *elapsed latency time* to a value that is the historical elapsed latency context for the particular Web server involved. Elapsed latency is the time from the invocation of Web fetch to the point at which rate becomes non-zero. After this, latency time remains static at its last calculated value. The historical context is calculated in a similar manner to elapsed time. However, latency historical context is associated with servers, so many different URLs might have the same historical context for latency.

Before defining rate, we describe another observable that we have identified: *completion* represents the progress of a fetch towards completion. It differs from other observables in that there is no historical context. Instead, the completion observable indicates the amount of a document that has been transferred at the moment off observation.

- The **completion** observable is defined as the ratio of current bytes transferred to the size of the Web resource in bytes. Thus, completion is zero until rate becomes non-zero, and cannot rise above 1.0.

We define rate in terms of completion:

- Persistent relative **transfer rate** is the ratio of a currently calculated rate to a single value that represents a historical context for previous invocations. The current rate is defined methematically as the slope (gradient) of the completion observable and since completion never decreases, rate is always a positive number. The calculated rate must be specified in the same units as the historical rate. We do not enforce any particular means for calculating historical rate, but typically, it would be calculated by an accumulation of average rates over download duration, requiring the number of invocations for that fetch to be stored. Rate is associated with particular Web servers so many different URLs might have the same historical context for rate.

In addition to the rate, time, latency, and completion observables, we have identified one more: *probability of success* uses the historical context for observables to allow reasoning about the likelihood of success for a Web fetch, in terms of its previously observed success rate. Querying a Web fetch for probability of success returns the ratio of the number of successful invocations to the total number of invocations. This gives a number between zero and one that is an approximation to probability of success for future executions. Later examples show that probability of success is a useful observable in conjunction with the others. For example, a low probability of success might indicate that a site is regularly offline, so in conjunction with a large observable latency, failure could be interpreted more readily.

- Persistent relative **probability of success** is defined as the ratio of the number of successful downloads to the number of attempts for a particular URL. There is no current observable as such, and the relative observable remains static across the duration of document fetch. The historical context contains only the number of invocations and number of completions, which are discrete quantities and so have no associated units of measurement.

In all the definitions so far, we have assumed that a persistent context actually exists for the URLs or servers from which we are downloading. However, since there will always be a first time for downloading a URL, we must consider the issue of first time transfer observables. On the first time downloading a URL, there is no persistent context from which to calculate ratios. Instead of seeding the persistent context with arbitrary absolute values, we instead choose to define the value of observables for first time transfers. We are interested in acceptability based on the norm, but there is no real notion of what is normal for a first time transfer. We must have the persistence mechanism do a little work:

- *Rate* – throughout the first time transfer, rate is calculated relative to the average rate seen so far for that transfer.
- *Time* – time is estimated according to the current rate of transfer and remaining amount of document to be downloaded.
- *Latency* – latency is calculated relative to the average network latency. This is the same as the network relative observable, defined later.
- *Probability* – 1.0 throughout duration.
- *Completion* – as normal.

Now that we understand how persistent relative observables are calculated, we can consider ways in which their values might be used in interpreting failure.

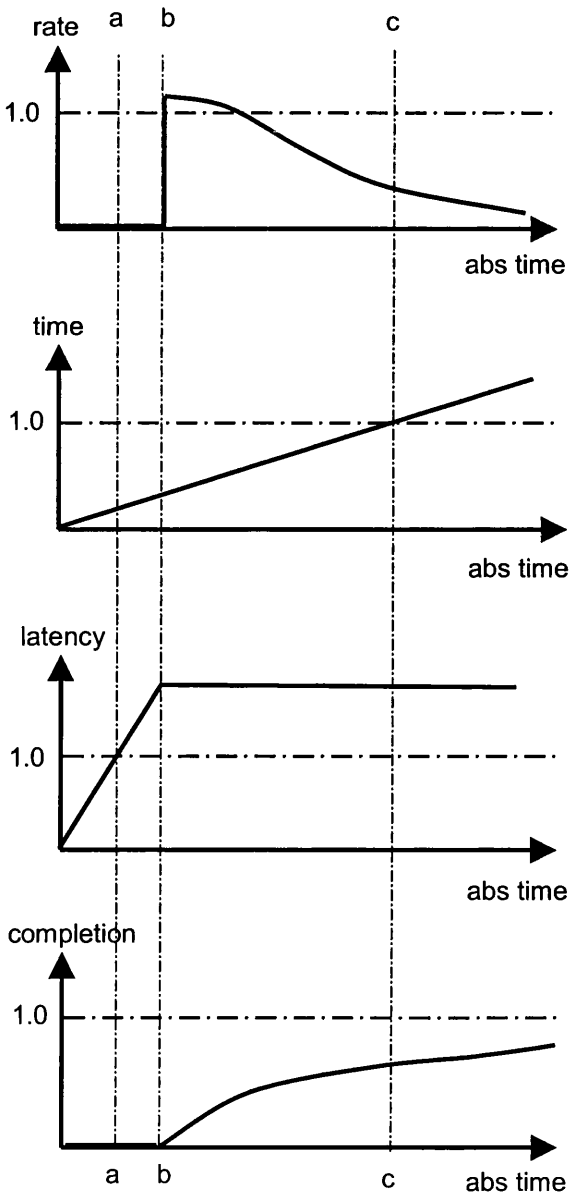
Flexibility in Interpreting Failure

For time, rate, and latency, *acceptable* progress for a Web fetch is indicated by a relative observable value close to 1.0. For rate, deviations either side of 1.0 indicate that a computation has an observable behaviour either better or worse than that 'which is to be expected', based on historical observation. For example, a Web transfer with an observed rate of less than 1.0 implies a slower transfer, whereas a rate greater than 1.0 indicates an increase in bandwidth. In general, any rate greater than 1.0 may be deemed acceptable, and values less than 1.0 cover a range of acceptability. A relative time greater than 1.0 indicates that a transfer is taking longer than before and the greater the value, the less acceptable. Any relative time less than 1.0 would typically be acceptable. Latency is similar to time in this context. Note that the concept of acceptability for a particular observable may be dependent

upon the value of other observables. For example, a high time observable is more acceptable if rate is also high.

There is a different notion of acceptability for the probability and completion observables, since they cannot exhibit values greater than one. Their observables are generally only useful when considered along with others. However, we consider the ability to arbitrarily correlate the values of different observables in our domain to be the primary source of flexible failure interpretation.

As a Web fetch proceeds in real time, its observables fluctuate. At a particular point in time, the current values of observables can be correlated to provide a holistic view of transfer progress. That is, a higher level interpretation of transfer progress can be formed from the union of observations than that possible given values for observables in isolation. For example, consider the four graphs adjacent that show the observable behaviour of a hypothetical Web transfer over absolute time. The curve for rate does not reflect a true transfer, since these tend to be highly variable (Chapter 2), but we use a smooth curve for simplicity. All four graphs have the same horizontal time axis, and each vertical broken line represents a moment in real time for which observations are particularly of interest, aiding visual correlation of observables. a is the average connect time taken for previous invocations of equivalent Web transfers. That is, it represents the absolute average latency time in the historical context of that transfer. b is the absolute latency time for the current transfer, and c is the historical completion time.



In the top graph, which shows relative rate across transfer duration, we see that initially, the transfer rate is greater

than that seen before, but soon falls below the historical average and gradually approaches zero. The second graph shows relative time linearly increasing. At absolute time c, relative time rises above 1.0, meaning that the transfer is taking longer than is perhaps to be expected. In the third graph, relative latency rises above 1.0 at time a, indicating that adverse network conditions are causing this transfer to suffer from greater latency than before. Latency continues to rise until it is approximately twice the historical average, at which point the transfer begins. During transfer relative latency is no longer a factor, and its value remains steady. The end of latency at point b correlates with the sudden jump in rate as the transfer begins, and with the ascent of completion percentage in the final graph. Completion percentage only begins to rise at this point because until latency ends, no data is being transferred.

Observation of rate or time in isolation might seem to indicate that interpreting failure would be reasonable, since rate has fallen to almost zero, and the fetch is taking longer than expected. However, when considered with completion percentage, it is apparent that despite the low transfer rate, its overall progress has been quite good since the download is near to completion. Given this, we might want to overlook the rate and extend the time constraint a little longer, for example.

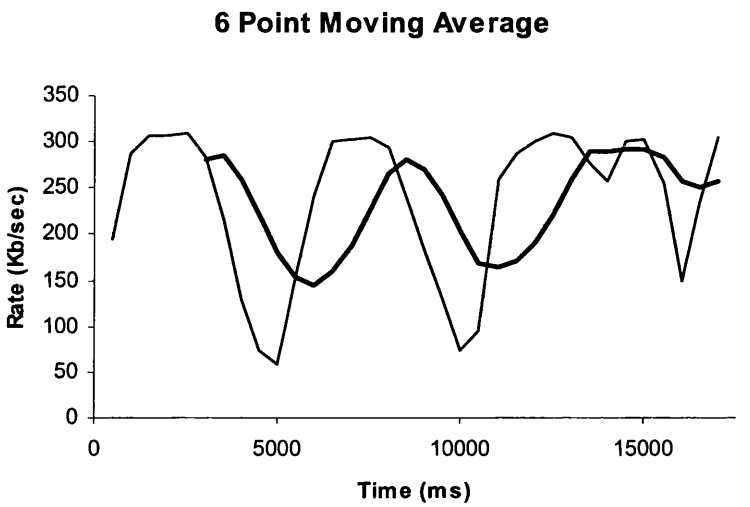
Diminishing the Impact of Rate Troughs

In Chapter 3, we discussed the issue of rate troughs relating to our implementation of the service combinator algebra [58]. Our implementation of service combinators incorporated persistent relative observables, so we discuss it here. Rather than constraining values expressed in terms of absolute measurements, the timeout and limit combinators accept relative values. We found persistent relative observables to be a useful addition to the service combinator algebra. However, experimenting with the implementation confirmed a suspicion that we alluded to in Chapter 2. Even with a dynamic rate window of 700ms, we found that failure would frequently be erroneously interpreted due to intermittent troughs in rate. On analysis we discovered that this failure interpretation would occur at times when overall transfer progress was good. That is, the troughs in rate are short enough to be of no consequence. Since Web fetches are naturally 'bursty', and particularly so under heavy network load (Chapter 2), we are motivated to find a way of diminishing the impact of inconsequential troughs in rate. Our method involves the use of a 'smoothing function', which reduces variability in dynamic rate.

The particular smoothing technique we used is based on a *moving average* technique. There are other potentially applicable techniques, such as splines, for example. However, moving averages are easy to calculate, and are particularly suitable for incremental calculation as data

is dynamically produced. The moving average technique produces smoothed data by averaging the last n actual data points, in this case dynamically calculated rate. As the i^{th} data point is calculated, the $i\text{-}n^{th}$ data point is no longer used in the calculation. Thus, the calculation of a moving average is based on a 'sliding window' technique. The graph below shows the actual dynamic rate and corresponding moving average (thick line) for a midday transfer from a UK academic site. We have found a six point moving average to be an effective compromise with a 700ms window granularity for rate calculation.

As shown in Chapter 2 – *Analysing Web Failure and Performance*, a smoothing effect can also be obtained by increasing the granularity of dynamic rate calculation. However, doing so reduces the actual number of rate observations for a transfer. We feel that applying a smoothing technique gives a better representation of dynamic rate.



Patterns of Human Failure Interpretation

Humans can interpret failure for Web fetches in many different ways. Persistent relative observables allow the automation of human-like interpretations. Following are several example human policies for failure interpretation, and descriptions of how they can be expressed in terms of persistent relative observables. The descriptions are in English, but we assume that they are implementable given even a simple programming language integrated with persistent relative observables. In the next chapter, we present concrete implementations.

Human: "Don't waste time waiting for connection to a site that is probably down."

Here, we wish to minimise the amount of time before taking remedial action such as retrial or access of an alternate resource. A low value for probability of success indicates that a

particular resource is frequently unreachable. Correlating this with connection time latency, we can save time by tightening acceptability limits on latency by an amount proportional to the inverse of probability. That is, the less likely a resource is available, the less time we are willing to wait in excess of 'normal' latency time (1.0). The correlation of latency and probability here allows failure interpretation to take place earlier, while retaining confidence that the interpretation is correct.

Human: "Accept intermittent transfer from heavily loaded servers."

As shown in Chapter 2, heavily loaded servers and network connections can be subject to patterns of 'burst' transfer as the server and routers struggle to achieve fair time sharing for all clients. Transfer rate can drop to zero for periods of several seconds under these circumstances. Failure interpretations based on rate can be unreliable in this context, since although overall transfer progress might be acceptable, a rate of zero even for a short period might be interpreted as failure. Although we have already discussed methods of minimising intermittent troughing, under some circumstances we may still wish to explicitly relax rate constraints. For example, some of our experiments suggest that that a higher than average latency indicates heavy server load, so under these circumstances we might allow violation of a rate constraint more than once before interpreting failure.

Human: "Invest more time in transfers that are close to completion."

Many Web servers allow site administrators to set policy for client priority. The graphs for the hypothetical Web fetch above exemplify possible behaviour for servers that reduce the priority of a transfer the longer it continues. This policy manifests itself as a gradual decrease in transfer rate while downloading large resources. Under these circumstances, failure interpretation becomes more likely towards the end of transfer. If failure is interpreted and remedial action taken, it is possible that download of alternate resources and in particular retrial of the same resource will exhibit similar behaviour. Towards the end of a transfer, it might make sense to invest more time in the hope that completion is forthcoming. To this end, rate and time can be correlated with completion percentage, so that rate and time constraints are relaxed if the resource is close to being completely downloaded. We might relax the rate and time constraints by a factor proportional to completion percentage.

Human: "Be pessimistic in the morning, expecting downloads to take longer."

In chapter 2, we demonstrated that network load increases at certain times of the day, for example early in the morning when people generally arrive at work. It can be useful to adjust constraints based on the time of day. We can adopt a pessimistic approach to download, by loosening all constraint values by a multiplicative factor if the time of day falls within a specified range.

Human: "Expect fetches with higher than average latency to take longer overall."

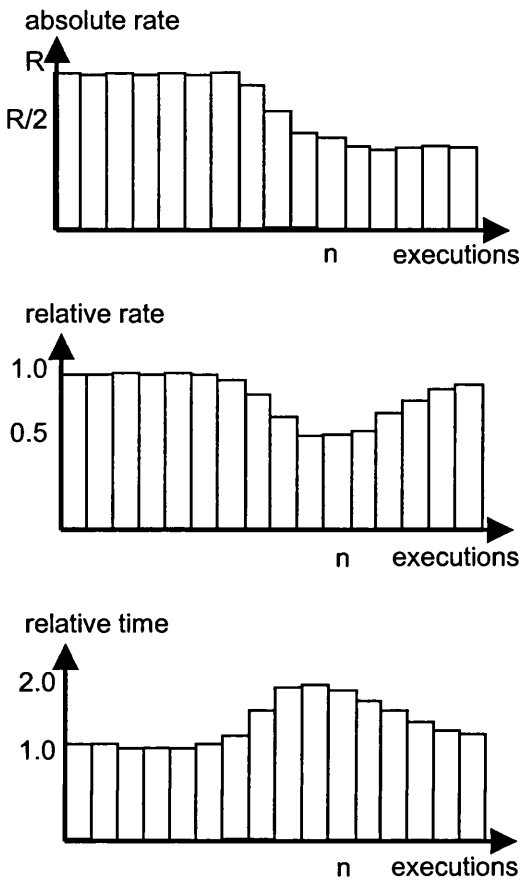
As shown in chapter 2, transfer rate is only partially dependent on geographical location. For example, rates for transfers between Australia and Europe are often comparable to transfers internal to Europe. However, latency is always a major factor in geographically distant transfers, especially if they are routed through satellite communication channels. If there is a significant latency overrun, this impacts the overall download time, and may result in premature timeout. For example, if a fetch takes five times longer to connect than usual, but its transfer rate is acceptable after that, then it makes sense to relax any overall timeout, since a significant portion of the overall timeout has been 'used up' by latency.

These policies for failure interpretation are only some of those that might be adopted by a human browser. Our conceptual domain is an effective basis for the expression of many more useful failure interpretation policies.

Persistence Properties of Observables

The graphs above show how observables might fluctuate across the duration of a single Web fetch. In addition to these short-term fluctuations, persistent relative observables for Web fetches fluctuate according to long- and medium-term trends in the network environment. The three graphs below show typical observable behaviour for a Web fetch over many executions. The vertical axes represent average observables over completed Web fetches. The horizontal axes represent a series of successful Web fetch invocations over time, and for simplicity, we assume that the time increments between them are the same at approximately one hour. Since our conceptual domain incorporates persistence, the Web fetches may be over several program invocations, and the historical context is maintained.

The top graph shows the absolute rate observed, on average, for many invocations of a particular Web fetch. The n^{th} fetch is observed to proceed, averaged across its duration, at half the previously seen rate. After that, however, the trend in rate begins climbing again, very slowly. Such observables are typical of a site in North America accessed from Europe, for example. That is, at mid-day in Europe a slow-down in Trans-Atlantic network traffic is observable as the workday begins in North America. The next graph shows how the average relative rate fluctuates, and should be compared against the absolute rate. The 'dip' corresponds to the halving of the absolute rate, and we see a relative rate of 0.5. However, as time goes on, the relative rate is seen to rise towards 1.0 because the lower absolute rate begins to influence the average of the persistent historical context for that Web fetch. In this way, the perception of rate adapts to changes in the network environment. The final graph shows similar adaptation of relative time. When the average rate of transfer halves, the average download time rises to approximately twice that seen previously, and this manifests itself as an observed relative time of 2.0. Like the relative rate, this adapts over time.



The persistence mechanism for our domain absorbs changes in the network environment. The rapidity with which changes are absorbed depends on factors such as the size of the history (window size) and how the history is calculated. We do not concretise our persistence mechanism in this respect, since various schemes may be of use in different situations. For example, an application that performs frequent access to a small set of sites might perform better with a small window size, in order to have the persistent context closely follow trends arising from the time of day.

The latency, completion, and probability observables behave in a similar manner to rate and time with respect to persistence. It is worth noting at this point that since the completion observable is intrinsically linked to document size, it immediately absorbs changes in size

resulting from document update, because the observable has no persistent context as such and depends only on current document size.

The persistent time observable for a document may be compromised by changes in document size. If the size of a document changes, so will the average time to download it. However, like changes in the network environment, the persistence mechanism is capable of absorbing them, assuming that changes in size are not radical. Future work is to analyse the patterns on the Web of changes in document size. If it is found that documents frequently change size, and the difference in size is often great, then it might be worthwhile addressing this fact directly in the mechanism. For example, the persistence mechanism could track document size, and if it changes adjust the time value in the persistent context by an amount proportional to the change in document size. Although such change makes an assumption about rate being consistent across transfer duration, a mechanism similar to this is likely to improve the reliability of the persistent context.

Generalisation of Observables to all Computation

Some of the observables we have identified are useful for non-deterministic computational operations other than Web fetch. For example, elapsed time and probability of success are meaningful for all atomic operations whose progress is non-deterministic. To bring other non-deterministic operations into our conceptual domain, we must find sensible default values for those observables that are not significant with respect to that type of operation. That is, observables of a non-deterministic operation that are not significant should be *deterministic*. A *deterministic operation* is a special case of this, in that *all* of its observables are deterministic. We see the incorporation of all computation within our conceptual domain as a logical generalisation, since it allows the domain to be specified in terms that are independent of the type of computation being performed. To do this, we need to generalise the concept of persistent relative observable to all deterministic atomic computational operations.

Deterministic operations have predictable observables. By 'deterministic operations' we mean an atomic unit of computation with deterministic progress. For example, it may be guaranteed to execute in unit time. In the Focus run-time system, described later, mathematical operations, string operations, and file system operations are all atomic, deterministic, and execute synchronously. However, these units of computation map to several underlying operations on the physical hardware, some of which may be asynchronous. In general, units of computation don't have totally deterministic progress on real machines due to memory caches and hard disc seek times, for example. However, we choose to ignore the issue of non-deterministic computational progress for synchronous units of computation in language run-time systems. These units of computation are executed on a closely coupled

architecture, and the time scales involved are orders of magnitude smaller than those of network access. We are not interested in the performance of closely coupled architectures. In essence, we split the instruction set of our programming language into two classes, deterministic operations and non-deterministic operations. Internal to the language run-time system, these two classes map directly to synchronous and asynchronous execution. That is, they do not block any other concurrently executing threads. All synchronous operations execute in 'unit time'. Asynchronous operations do not execute in unit time, and may or may not execute in a predictable time.

Failure of non-deterministic operations is inferred by deviation in observables from what is considered 'normal'. Thus, observables for deterministic operations should be predictably normal, given the intuition that it is not generally useful to interpret failure for deterministic operations within the context of our conceptual domain. For example, the transfer rate in Kb/sec of a subtraction operation is not particularly meaningful, and is difficult to define. For Web fetches, which are non-deterministic, persistent relative observables fluctuate in a pattern around the number one, which along with zero is a *grounding* value. An observable of one indicates normal computational progress, or more accurately, *acceptable* computational progress. Thus, we choose the number one as the default observable value for deterministic operations. However, we do not enforce this indiscriminately. For non-deterministic operations other than Web fetch, observables that are not meaningful adopt the default value. Meaningful observables retain the semantics defined for Web fetches.

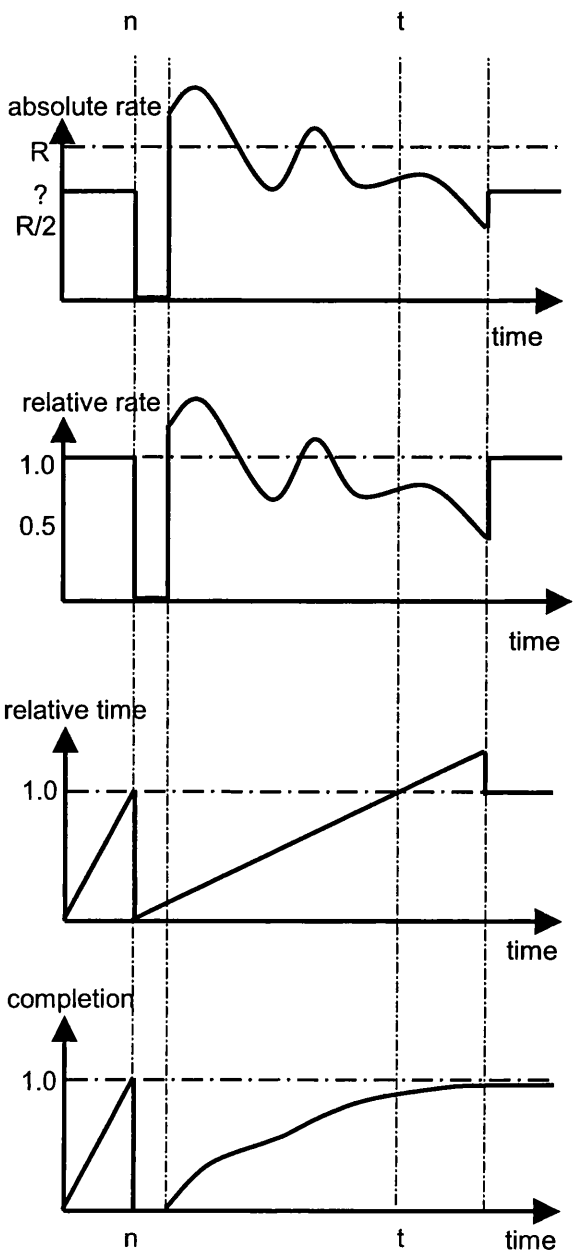
Given that in our domain, all operations export the same set of observables, we can reason with single operations and sequences of operations (aggregate computations) in the same way. As an aggregate computation proceeds, its observables are available seamlessly between the completion of one computation and the execution of the next. In this way, we abstract over the actual nature of computation taking place, whether they are primitive operations such as string concatenation, or more abstract sequences. However, the granularity of persistence for relative observables is at the operation level. This means that at any given moment, the persistent relative observable of an aggregate computation is actually the persistent relative observable of the atomic operation currently being executed.

Consider the following program fragment, in which a Web fetch is preceded by a sleep operation, and followed by a deterministic string computation. Sleep here is an example of how we can add a new operation to the conceptual domain and map persistent relative observables onto it in a meaningful way. Although it can be argued that sleep is a deterministic operation we treat it as non-deterministic because internal to the language run-time system, sleep must be executed asynchronously. This allows the observables to change

throughout its execution. The graphs below are an example of how the values of observables might fluctuate over the execution of the aggregate computation.

```
sleep n
let html = get ("http://www.protean.org/ ")
processHTML(html)
```

In the first graph, we observe the absolute rate of the computation as time passes. R is the average absolute rate that has been seen before for the Web fetch, in bytes per-second, say. The actual value of R is of little concern; more interesting is the relative rate of the current transfer with respect to R. Moreover, we do not wish to define the absolute value of rate for deterministic operations. Thus, our domain abstracts over these absolute values, and they are never directly observable. The second graph, shows that the curve for relative rate follows that of the absolute rate, but here we have a more sensible unit of measurement for the vertical axis since we can obtain a rate value for the deterministic portions of the aggregate computation. *Sleep*ing has no meaningful absolute rate, and so we define it's relative rate as one. However, we define its time and completion observables to increase monotonically (never decreasing) towards one. After *n* units of time have passed, relative rate drops sharply to zero as the Web fetch begins and experiences connection latency. Once the connection is made, data begins to transfer and rate is seen to fluctuate. We assume that the progress of the fetch is acceptable, and it



completes normally, after which the deterministic string operation is invoked and all observables snap to the number one.

Given the set of available observables, program logic must dictate whether or not to interpret failure. Typically, this program logic is applied for the duration of the entire aggregate computation, since we have abstracted over the nature of computation and cannot directly determine what operation is being performed. However, since deterministic operations never have observables that deviate from what is considered acceptable, failure interpretation will not occur while deterministic portions of the aggregate computation are executing. This means that we can apply failure interpretation logic uniformly across a computation, secure in the knowledge that failure will only be interpreted during the non-deterministic portions of the aggregate computation.

The persistent relative observables domain alone is not sufficient to define a programming language. However, because we have defined the concept of observable for all computation, it can be easily unified with other conceptual domains in the construction of Web programming languages. Moreover, by identifying generalised observables within a distinct conceptual domain, we do not limit the means by which they can be reasoned about at a higher level. That is, the domain is designed to allow failure interpretation independent of any particular means for specifying remedial action once failure has occurred; the design of which is left to language implementers.

Observables and Concurrency

Concurrency is important in any system that has large IO overhead, since it allows the processor to be utilised during periods it would otherwise be idle, waiting for relatively slow hardware. Accesses to the global network are subject to unpredictably high levels of overhead in the order of seconds, so any programming language designed for computation in this domain should incorporate concurrency. Concurrency is a concept introduced within the computational model of a language, and not its conceptual domain, so we do not deal with it directly here. However, we assume that any language incorporating our conceptual domain will support concurrency, and so we must ensure that the properties of persistent relative observables are in concordance with concurrency. Moreover, to improve the effectiveness and flexibility of our domain, we wish to identify any interplay between observables and concurrency, and attempt to incorporate concepts within our domain that are useful in a concurrent context.

Aggregate computations are sequential. Issues such as concurrent update by computation and synchronisation are beyond the scope of our domain, as they should be dealt with by the programming language's concurrent computational model. Our domain has no properties that

preclude the concurrent execution of aggregate computations, and observations of individual computations can take place independently. The only aspect of our conceptual domain that potentially interferes with concurrency is the persistence mechanism. On successful download of a resource, the associated persistent historical context is updated. In a concurrent context, there may be consistency issues with respect to this update. Although we do not see them as being significant, we avoid the problem by demanding that individual updates of the persistent historical context be atomic transactions. This ensures that the historical context can never be observed in an inconsistent state.

Concurrency allows synchronous Web fetches to be made in an asynchronous context, with corresponding improvements in processor usage and program efficiency. Efficiency aside, concurrency is valuable in its flexibility as a structuring tool for computations. For example, human browsers often invoke several simultaneous Web fetches. In some cases, two or more of these fetches may be co-dependent in some way. Large files are often split into several component archives, which can be downloaded concurrently. Success of the overall download is dependent upon successful download of all components. Another form of concurrency employed by human browsers is that of alternate download. For example, a human browser wishing to read the days news headlines might attempt to fetch the CNN homepage. If the fetch is not achieving acceptable progress, then the human browser might invoke a fetch of the MSNBC homepage concurrent with the CNN fetch. The human browser reads the content of whichever fetch completes first. In this way, human browsers adopt a form of concurrent scheduling policy, based on observations of progress. Such scheduling policies may be based on comparative observations. For example, pessimistic human browsers might concurrently invoke fetches of CNN and MSNBC from the outset, and terminate the slower of the two. We class the general form of this behaviour as *alternate computation*, where the results of two or more computations are equally valid, and whichever completes first forms the result of the overall concurrent computation.

There is a need for comparison of absolute observables. However, in our conceptual domain, observables for a transfer are calculated relative to a historical context for that transfer. This means that a comparison between the persistent relative observables of two distinct fetches is not particularly meaningful. For example, consider a concurrent download, invoked from Europe, of two equivalent documents U and J from the USA and Japan respectively. J has an absolute rate of 10Kb/sec and a historical rate context of 5Kb/sec, whereas U has a rate of 20Kb/sec and a historical context of 40Kb/sec. These absolute rates result in relative rates of 2.0 and 0.5 respectively. An observer deciding to terminate U on the basis of a comparison between relative rates is actually terminating the transfer most likely to complete first. This

example shows that comparisons of persistent relative observables have no relationship to comparisons of absolute observables.

Our conceptual domain elides absolute measurements and so removes classes of error in programs by making them location and hardware independent. Moreover, the generalisation of our conceptual domain to all computation is critically dependent on the fact that we mask the absolute units of measurement for observables associated with deterministic computation. However, since absolute measurements are useful in many situations, it is unreasonable to prevent programmers from using them entirely. For this reason, we have developed an extension to our conceptual domain we term *network relativity*, which approximates absolute units of measurement without compromising their secretion with respect to deterministic computation.

A network relative observable allows the observation of progress for computation in terms relative to the average progress of all computation. For example, the network relative rate observable for a Web fetch is calculated by taking the ratio of current dynamic rate to the average of *all* persistent rates stored. If the average download rate for all transfers is 10Kb/sec, and the dynamic rate of a Web fetch is 7Kb/sec, then the network relative rate for that fetch is 0.7. Network relativity allows meaningful comparisons between the observables of different computations (fetches). The concept applies to each class of observable as follows.

- *Rate* – ratio of dynamic rate to average rate of all fetches. This is a directly meaningful approximation to absolute rate.
- *Probability* – How frequently this download succeeds compared with others.
- *Latency* – returns value relative to average network latency.
- *Completion* – network relative is not meaningful for completion, so we do not include it as network relative.
- *Time* – network relative is not meaningful since download time is related to document size.
- *Deterministic computations* – always one, since is calculated by dynamic observable ratio with historical context for all deterministic computations, which is one.

Like persistent relative observables, network relative observables are independent of absolute units of measurement. They are particularly useful for comparative analysis of observables for different computations.

Summary and Analysis

We have described a conceptual domain providing persistent relative observables, which is an abstraction with the following properties:

- It gives language designers the means to provide constructs that allow flexible failure interpretation in terms of relationships between constraints on observables.
- The separation of observables as a distinct concept allows the implementation of arbitrary mechanisms that control program flow based on a high-level interpretation of fluctuation in observables.
- Languages incorporating the conceptual domain can provide abstractions that capture the notion of 'acceptability based on the norm' without overhead.
- The persistent context of programs automatically adapts to changes in the network environment, making them future proof.
- Programs are portable and mobile, since values associated with observables are expressed in terms independent of absolute units of absolute measurement.
- Persistent relative observables allow genericity in control, since for all computation observable values are constrained within a pattern of fluctuation around a single number, 1.0.
- Generalisation of observables to all computation simplifies unification between our conceptual domain and traditional conceptual domains, since it allows failure interpretation logic to be applied uniformly.

Traditional programming languages do not generally include directly the concepts of rate, latency, completion percentage, or probability of success because they are not present in their conceptual domains. This places a burden on the programmer who wishes to implement failure interpretation and control based on these concepts, since the language's computational model may be inappropriate. By providing the concept of observables orthogonally within a conceptual domain, we encourage language designers to provide abstractions that are appropriate to computation in the Web domain.

The only notion relevant to Web programming provided by traditional conceptual domains is that of time. An indirect consequence of this is that traditional programming abstractions tend to place a heavy reliance on timeout as a means for interpreting failure. We have argued that timeout as a means for failure interpretation is inflexible, inefficient, and unreliable. Even without any exercises in comparative programming, it is self evident that each property of our conceptual domain listed above is provided in a more direct manner than is possible with traditional conceptual domains and languages. This eases the design and implementation of Web programming languages and abstractions. Whether these properties are useful for Web programming itself is a separate hypothesis that we address in the next chapter by designing a programming language that incorporates the observables conceptual domain. We intend to show that persistent relative observables can be incorporated into a high-level programming language that successfully integrates the concepts of concurrency, flow control for failure, and flexible failure interpretation in the Web domain.

6: Observation and Control with Supervisors

In this chapter, we describe the *supervisor* programming abstraction, which is intended to allow effective computation over the persistent relative observables conceptual domain. The primary design goal of supervisors is to achieve clean integration of concurrency, flexible failure interpretation, and flow of control for failure. Supervisors are abstractions over concurrency that are syntactically similar to functions, but are intended for the specification of failure semantics in Web computations. In essence, supervisors monitor and control concurrent computations passed to them as parameters. At run time, each supervisor corresponds to a *distinguished thread.* This thread is responsible for interpreting the conditions of failure (or other circumstances that require intervention) in the monitored computations, and directing concurrent flow of control for appropriate action.

Research by Randell [63] indicates that programming abstractions for failure prone computation should allow the syntactic and semantic separation of control logic from computational logic. Web computation is inherently failure prone, and the design of the supervisor abstraction reflects this by enforcing the strict separation of computational logic and control logic. With supervisors, no cooperation or communication in general computation is possible between the distinguished thread and the threads it supervises. The supervisor abstraction is modular, since the internal logic of supervisors and supervised computations is mutually hidden.

Whatever the activity of a computation, information about its behaviour is observable in its set of persistent relative observables. The values of observables are constrained within a pattern of fluctuation around a single number, 1.0. All computations export the same observables, and many exhibit similar patterns of fluctuation in observables. This means that the same supervisor can control a broad class of computations, giving a degree of genericity to supervisors. Similarly, many different supervisors may be applicable to a particular computation, engendering different failure interpretation and control flow characteristics in each case. Observables are the only means by which supervisors can obtain information about executing computations. Thus, the development of supervisors is not directly related to knowledge of the program logic for supervised computations, but is based on patterns of expected fluctuations in observables.

Once failure has been interpreted by whatever means, supervisors provide support for *automated backward error recovery.* In essence, backward error recovery is concerned with returning the system to a previously known reliable state, before the occurrence of failure. This differs from forward error recovery, which is concerned with 'handling' the failure and repairing state, rather than returning to a previous state. Exception handling mechanisms are

designed for forward error recovery and recovery blocks provide automated backward error recovery. We describe recovery blocks in Chapter 7 – *Analysis of Related Work*, and exception handling in Chapter 8. Research indicates that manual repair of erroneous state is difficult since it places a large cognitive burden on the programmer [94]. This supports the idea that backward rather than forward error recovery is more appropriate when computing in a failure-prone non-deterministic context. Failure is non-deterministic and common on the Web, and in Web programs there can be an arbitrary amount of state made erroneous by computation dependent on network connections that fail. Thus, automated backward error recovery is perhaps most appropriate in this context.

Focus

Supervisors can be embedded in any language that incorporates the persistent relative observables conceptual domain. However, we present a concrete programming language called *Focus* (Flow Of Control Using Supervisors), in which supervisors are the sole construct for concurrency. Focus is an imperative programming language designed to allow the expression of computations over the Web in a concise, flexible, and easily understood manner. It is an experimental language, primarily intended as a vehicle for the supervisor abstraction. Although the language model is designed to integrate well with supervisors, we do not assert that its structure is either ideal or necessary for them. Indeed, we intend that supervisors be amenable to integration with a broad range of different languages, including object-oriented and functional languages.

Focus is a simple language that contains minimal clutter, so that we can present the supervisor concept clearly. However, in the presentation of some examples assume the existence of language concepts not present in Focus, without explanation of how they might fit within the language model. In particular, we refer to parametric polymorphism [64]. Supervisors are independent of polymorphism, but its presence affords extra flexibility as they would to any programming language. There are no conceptual barriers to augmenting Focus with mechanisms for polymorphism.

Focus has the following properties, which are generally thought to be beneficial in any high-level language:

- *Strongly and statically typed* – all type checking takes place at compile time, and the compiler and run-time system together ensure that every execution is sound in that it has a defined meaning.

- *First class values* – values of all types have the full range of applicability normally granted to simple types such as integers, say. That is, all values, including function and supervisor values, can be bound to identifiers, assigned to locations, and be passed to and form the result of functions and supervisors [65].
- *Orthogonal* – there are no arbitrary restrictions on the applicability or composability of operations or data types, language features are independent, combine in regular ways, and uniform syntax is applied wherever possible.
- *Block structured* – static nesting of scope for arbitrary sequences of commands. Blocks may have values and so are valid anywhere that an expression is.
- *Explicit locations* – values that are mutable must be explicitly declared as such. Moreover, locations must be explicitly dereferenced, distinguishing between l-values and r-values in program text.
- *Automatic memory management* – a language with no concept of deletion removes a major burden of complexity from programmers. Focus automatically retains all reachable values.

Focus is a Web programming language, and incorporates fundamental concepts to this end:

- *Atomic Web fetch operation* – the Focus model of download replaces the notion of a transfer stream with an atomic fetch, and documents download completely or not at all.
- *File system unified with locations* – after declaration, files are indistinguishable from locations, and are updated in the same way. The motivation behind this abstraction will become clearer when we describe the supervisor mechanism for controlling concurrent update.

A more detailed description of focus is presented in the appendix, including a formal BNF syntax definition, and a description of its implementation.

Supervisors

Supervisors are *concurrency constructors* with associated program logic. They are syntactically similar to functions in that they are parameterised by arbitrary expressions, which in turn are bound to formal parameters within an expression that is the *body* of the supervisor. This implicitly separates the *control logic* in the supervisor body from

computational logic in the parameter expressions. The parameter passing mechanism for supervisors is that the parameter expressions are evaluated *asynchronously*. Expressions passed as parameters to a supervisor are not evaluated eagerly before supervisor invocation. Instead, the evaluations of the parameter expressions begin concurrently, with each other and with execution of the supervisor body.

A view of the parameter expressions as their dynamic evaluations, and not the value they compute, gives rise to entities we term as *threads*, and the dynamic view of the supervisor body we term as the *distinguished thread.* In the previous chapter, we defined an *aggregate computation* as a computation composed serially from other computations or atomic operations. An aggregate computation is abstract in that it is not possible to determine the nature of computation taking place other than by interpreting fluctuation in its observables. Each thread in the supervisor model is an aggregate computation as defined in the observables conceptual domain, but supervisors allow them to exist in a concurrent context. The Focus run-time system incorporates a concurrent threading environment compatible with the observables conceptual domain. However, supervisors abstract over this threading environment and Focus provides no other thread constructor. In Focus, passing parameters to a supervisor is the only means of creating concurrency, and threads can only be observed in the context of a supervisor body.

When an expression is specified as an actual parameter to a supervisor, it is implicitly used in the construction of a thread. The supervisor body can specify observations of its named parameter threads with special functions that correspond to the set of available persistent relative observables. With program logic, it can drive the scheduling of threads by interpreting the significance of these observations. However, the binding to a formal parameter has two interpretations: as a thread and as the value that it computes. The use of a formal parameter binding refers to a thread only if the context demands a thread type, for example when passed as a parameter to a thread control or observation function. Otherwise, the binding refers to a value and the distinguished thread blocks until the result of the parameter thread becomes available. The type of the value that each parameter thread computes is statically known, and this is checked for compatibility in the context of identifier usage.

Consistent with the function view of supervisors, it is the value computed by the supervisor body that forms the result of the overall concurrent computation. The results of computations passed to the supervisor are available, and typically the supervisor result will be based upon these. That is, the results of some or all of the concurrent computations are amalgamated in the supervisor body in order to form the result of the supervisor. Since the supervisor body is composed of arbitrary program logic, it may perform any amount of supplementary

computation in order to achieve this amalgamation. Moreover, it can base flow of control decision on the results of its parameter threads, for itself, or for other parameter threads.

The supervisor body can effect concurrent flow of control according to any state that is deducible from observing its parameter computations or obtaining their results, and is not restricted solely to the interpretation of failure. However, we see the detection and handling of dynamic failure conditions in parameter computations to be the major role for supervisors. For example, programmed failure semantics in a supervisor body might specify that the rate of computation for a particular parameter thread should not fall below a specified value; if it does, then the result of an alternative parameter thread is returned instead.

Thread observation and control

Despite the absolute control supervisors have over threads, they have no direct knowledge of the semantics of computation taking place within an observed thread. Consistent with the definition of aggregate computation in the observables conceptual domain, they cannot base their interpretation of thread's progress on anything other than the set of observables exported by that thread. However, we assume that supervisors will be developed in conjunction with the class of computations they are to monitor and control. This means that each supervisor will have programmed within it an underlying knowledge of the high-level semantics (failure or otherwise) that may be implied from a particular pattern of thread observables. Since at run-time the supervisor body is itself a thread, it may be observed and controlled by a supervisor higher up the dynamic invocation chain. Nested supervisor invocation is dealt with in a later section.

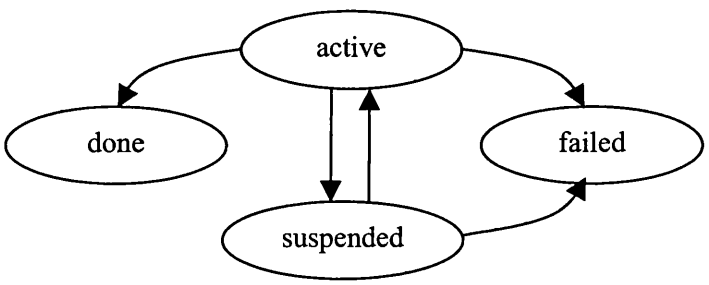
The primary 'observation' that the supervisor body can make of a thread is that of its result value. Any time a formal parameter is used in a non-thread context, this refers to the value that the thread computes and the supervisor body blocks until the value becomes available. In addition to the result value, we define observation functions that map directly to the observables in our conceptual domain. These are rate, time, latency, completion, and prob (probability of success). Each takes a thread parameter, and returns a non-negative floating-point number. When invoked, these functions return the persistent relative observable for the computation currently being executed by the interrogated thread. We define four observable thread *states*, and several related *control functions*. The supervisor body may suspend, activate, and retry[1] parameter threads. In addition, any thread may invoke suspend without parameters to suspend itself, which is useful for synchronisation. Threads can be in an *active* or *suspended* state, and we define the active and suspended observation functions, which

[1] Thread retrial is discussed separately in a later section.

given a thread return either true or false depending on the thread state. A suspended thread makes no computational progress until activated again. Suspending an already suspended thread or activating an active thread has no effect.

In addition to the active and suspended states, threads can be *failed* or *done*, and the operations failed and done are provided to query this state. Done threads have completed normally, and their result may be obtained without blocking execution of the supervisor body. Failed threads have either attempted to execute an operation that cannot be completed, for example in the case of a Web document not found or a divide by zero, or have explicitly invoked their own failure with the fail command. An attempt to obtain the result of a failed thread causes failure of the supervisor body. Failure is automatically propagated through all static scopes and the dynamic function invocation chain, and stops only when it reaches a supervised thread. We return to failure propagation in a later section, when we discuss nested supervisor invocations.

All thread states are mutually exclusive, but the relationships between them are captured in the state transition diagram below.



Although the observation and control functions require a thread parameter, there is no way for programmers to statically denote a thread type. Thread creation is implicit on passing an expression to a supervisor as an actual parameter. Active threads cannot escape the context of a supervisor body by assignment, return, or passing to a function, since they are always forced into a value context by anything other than an observation or control function. Thus, thread observation and control functions cannot be invoked outside the context of a supervisor body, since they cannot be typed.

Examples and Discussion

In this section, we present some small instructive examples. More 'real-world' examples are presented later in this Chapter. In the examples throughout, important sections of code are highlighted in bold (as opposed to the convention of highlighting keywords). The following short example shows how supervisors can amalgamate the results of the concurrent computations they supervise.

```
let combine = supervisor( a:int, b:int → int ) is a + b
```

Syntactically, this supervisor resembles a function. However, a and b are not values but computations, which execute concurrent with each other and with the supervisor body. The identifiers a and b in the supervisor body are in a value context, so the body blocks until the results become available. It then returns the value obtained from adding the results of the two computations. Blocking on the completion of a failed thread causes failure, so if either parameter computation fails, the supervisor body will propagate this failure to its invoker.

The next example shows a supervisor basing control flow decisions on observations of the computational progress of its parameter threads. waitfor is a construct that blocks execution of a thread until the associated boolean expression becomes true. Although waitfor can be used anywhere, we expect its primary role to be by the supervisor body. There are timing guarantees associated with waitfor that ensure the rescheduling of the thread within a known finite time after the expression becomes true. There are also timing guarantees with respect to the frequency of waitfor expression evaluation. The Focus run time system can be parameterised in this respect, since the actual time guarantees may be system specific. However, Focus provides reasonable defaults. Failure to meet any of these guarantees causes failure of the Focus run time system. In pragmatic terms, programmers need not be concerned with this. The run time system guarantees exist primarily because of the theoretical demands on concurrent programming systems with regard to computational progress and process scheduling.

```
let alternative = supervisor( a:t, b:t → t ) is {
        waitfor done a or done b
        if done a then a else b        //last a and b are in value context
    }

let html = alternative( get "http://hostA.org/", get "http://hostB.org/" )
```

In this example, we define a supervisor that provides the semantics of alternative computation, where either computation is a valid result, and the value returned is that of the first computation to complete. The supervisor is invoked to download two documents concurrently, though the parameter computations may be arbitrary. The body of the supervisor first synchronises its execution by waiting for completion of either thread before

continuing. In the waitfor conditional, passing a and b to done implies a thread context. The result of whichever thread completes first is returned as the value of the concurrent computation. The value computed by a thread is obtained by naming it in a non-thread context, and the supervisor body will block until the result becomes available. In this case no such blocking will occur, because the supervisor body has already synchronised on the completion of one of the threads.

The next example shows how observables can be used to interpret failure and direct the flow of control for concurrent computations. We pass constraint values for rate and time as parameters to the supervisor on invocation. Since parameter computations may be arbitrarily complex, passing a simple expression with no side effect is an approximation to parameterisation by value, since the observable properties of its evaluation are neither required nor significant.

```
let priorityAlt = supervisor( pri:t, sec:t, maxTime:float, minRate:float → t) is {
        waitfor   (rate pri < minRate and completion > 0)
                  or time pri > maxTime
                  or done pri
        if done pri then pri else sec
}
let html = priorityAlt( get "http://...", "error", 2.0, 0.33 )
```

Here, the body of the supervisor contains logic to provide asymmetric weighting favourable to the computation passed as the pri (primary) thread parameter. This is achieved by considering the sec (secondary) computation only if the computational rate of the primary falls below a certain level – in this case one third of that in the historical context, or if it takes twice as much time as it did on previous occasions. For document transfers, rate and time are related since a document that takes twice as long to download as before has on average half the rate across its duration as before. However, by constraining the *dynamically* observable rate in addition to time, we can interpret failure more readily, according to fluctuations not observable given only an average rate across fetch duration.

The values with which we constrain rate and time are passed as parameter computations to the supervisor. We use the results of these computations to determine how the supervisor body interprets failure. Thus, although they are computations, naming maxTime and minRate only in a value context means that they are effectively value parameters.

In the example, waitfor observes rate and time, and waits until either the constraint on time or on rate is violated, or if the overall computation completes. Because rate is zero during the latency phase of a fetch, we must check completion to ensure that we do not interpret rate failure erroneously during the latency phase.

Implicitly, if absolute failure of a thread occurs, its rate becomes zero, consistent with non-termination semantics for failure. Thus, absolute thread failure, for example due to divide by zero, need not be checked for the primary since it is captured by the observation of rate. Thread rate also becomes zero on thread completion. Thus, the check for not done in the third waitfor is redundant if we assume that minRate is positive, since it is captured by the rate constraint. However, we include the check for program clarity. The assumption of a non-negative minRate is reasonable since observable rate can never be negative.

The last statement in the supervisor is a conditional that determines whether the primary computation has succeeded or been interpreted as failed. If failure is interpreted, we return the result of the secondary, blocking the supervisor body if the secondary has not yet completed.

Side effect and thread communication

The threads involved in a concurrent computation need not be independent. For example, one thread may have to wait for the results of another to become available before it can continue. When a thread requires a series of partial results that are produced by another concurrent executing thread, there is a requirement for a means of communication between them. The simplest method of communication between concurrent threads is by update to shared memory locations. However, program consistency can be compromised by the fact that thread scheduling is non-deterministic. For example, one thread might attempt to read data that another has only partially written. Some level of abstraction is required.

Traditionally, there are two main approaches to abstracting over concurrency communication: *mutual exclusion* and *message passing*. The mutual exclusion approach avoids consistency problems arising from concurrent access to variables by allowing the specification of *critical regions*, in which only one thread can be executing at any time. The two main abstractions for creating critical regions are *semaphores* and *monitors*. For example, Java provides semaphores at the language level, and Modula-2 provides monitors. Message passing is suited to both distributed and locally concurrent computation, and involves the communication of information along explicitly created channels, or via an interface similar to function invocation. Message passing can be synchronous or asynchronous. Synchronous communication requires that all involved computations synchronise their execution at a specific point before they communicate, whereas asynchronous communication does not. Occam (which is based on CSP) and Ada use synchronous message passing.

With supervisors, we decided to use a shared memory model, since we consider message based models as too heavyweight for our lightweight threading mechanism. However, we have taken an entirely different approach to the standard approaches of monitors and semaphores. With these mechanisms, obtaining a lock on location access is the responsibility of individual computations. This places some of the burden of concurrency control on the computation itself. A design goal of the supervisor mechanism is that all control should be the responsibility of the supervisor.

In addition to making control the sole responsibility of the supervisor, we require support for automated backward error recovery. Over the years, research into coordinated (or concurrent) atomic actions [66] has been motivated by the requirement to roll back the computation of failed processes that are taking part in concurrent computations. As concurrent computations proceed, information flows between the individual processes involved and the number of dependencies grows. According to Randell, keeping track of these dependencies and undoing updates is a complex task [63]. We return to the issue of coordinated rollback in the next Chapter.

We have designed a mechanism that is based on a change of perspective with respect to undoing computation. Instead of allowing computation to be undone when threads fail, our mechanism takes a more pessimistic approach. This is borne out of the observation that Web computations are prone to frequent failure. With our mechanism, the computational effect of a particular thread is encapsulated, and not visible to any other thread unless its supervisor explicitly exposes that effect. Without explicit exposure, the computational effect of a thread is *implicitly discarded.* Thus, automated backward error recovery is achieved simply by abandoning a thread. No explicit action is required. In the next section, we describe this mechanism, the supervisor *environment model.*

Environments

In addition to its return value, supervisors permit side effect, either by the distinguished thread, or by parameter threads. In this section, we describe how supervisors can explicitly control this side effect with its *environment mechanism,* in order to allow computational interaction between parameter threads.

Environments are logical duplicates of the system store that are associated with every individual thread. Within each environment, mutable locations contain values valid with respect to that particular thread. On supervisor invocation, each nascent parameter thread captures an environment 'snapshot' viewed from the point of supervisor invocation. The original environment is the *parent environment,* and the capture is a *child environment.* The thread that is the supervisor body also captures an environment in this way. The child

environments with which expressions passed to a supervisor are evaluated, and the child environment of the supervisor body, are all isolated from each other. All newly created threads thus have their own environment to execute over that is distinct from the parent environment associated with the calling point.

Within each distinct environment, thread updates to mutable locations are not observable externally. This enforces mutual exclusion of *all* concurrent thread computation, since although concurrent threads may share portions of their namespace, identical mutable locations will exist independently, 'shadowed' within each captured environment. To allow cooperation between concurrent threads, we define an expose operation. This operation is parameterised by thread, and so can only be invoked by a supervisor body. Exposure reconciles the child environment of a parameter thread and the environment of the thread that invoked the supervisor, its parent. It causes the thread to propagate all updates it has made to the parent environment, and then recaptures that environment. In other words, any locations 'dirtied' by thread update first overwrite those in the parent environment, and then the thread again captures the parent. This has the effect of unifying the two environments, with updates made by the thread overriding any interim changes in the parent environment.

Since only the supervisor body can name parameter threads and type an invocation of the expose operation, only the supervisor body can cause a thread to be exposed to the parent environment. Therefore, only the supervisor can effect synchronisation of parameter thread communication, by exposing two or more threads to the parent environment, thus unifying the environments of threads that have updated shared locations[1]. However, the distinguished thread of the supervisor body cannot communicate with parameter threads in this way (cannot observe thread update) since it cannot name itself in an expose operation. This ensures mutual exclusion of control computation in the distinguished thread and algorithmic computation in the parameter threads. In order that the supervisor body may have an overall side effect, any updates made by the distinguished thread are automatically exposed to the parent environment when the thread completes. The supervisor body can return while parameter threads are still executing. Such parameter threads can have no further influence on the system, since they cannot be exposed[2]. The expose operation is valid at all times, be the thread failed, done, active, or inactive.

The example below defines a supervisor that has the semantics of parallel computation, with the results of the parameter computations being produced by side effecting update.

---

[1] The order in which the expose operations are applied is significant.

[2] There is an exception to this in the case of nested supervisor invocations. This will be dealt with later.

```
let par = supervisor( a:s, b:t ) is {
        while not done a and not done b do if failed a or failed b do fail
        expose a
        expose b                    //b updates take priority
}                                   //supervisor body is void
let x = loc(“”); let y = loc(“”)    //locations for side effect initialised to null strings
par( { x := get "http://hostA.org/" }, { y := get "http://hostB.org/" } )
```

In this example, the invocation of the supervisor specifies the concurrent download of two documents and assignment of the results to two distinct locations. The side effects of both computations are unified by the supervisor's invocation of expose, updates being propagated to the store at the level of the supervisor invocation. Notice how the supervisor requires no knowledge of the nature of the update. In this case, the locations are not even in scope for the supervisor body. a and b potentially share portions of their namespace, and so can update the same locations. For update made by a and b to the same location, those made by b will override since b is exposed after a.

The Hippo Core Language (HCL) [67], is the precursor to Focus. The fundamental primitive in HCL is a concurrency constructor for *alternate* computation, which was briefly described in the previous chapter. To recapitulate, alternate computation is bilateral concurrent computation, the result of which is exclusively that of whichever computation completes first. The mutually exclusive effect of computation is in terms of both the return value and in side effect of each part. Consider the example program below, which uses an alt combinator to invoke alternate computation.

```
let foo = loc("")

let result = alt(  { foo := "A"; get "http://hostA.org/doc.html" },
                   { foo := "B"; get "http://hostB.org/doc.html" } )
```

Here, only one of the updates to foo, “A” or “B” will occur, and the corresponding html document bound to result. If both computations fail, no update occurs, and failure is propagated instead. Generalised alternate computation can be implemented with the supervisor construct.

```
let alt = supervisor( a:t, b:t → t) is {
        while ((not done a) or failed a) and ((not done b) or failed b) do
            if failed a and failed b do fail
        if failed a or done b then { let res = b; expose b; res }
        else { let res = a; expose a; res }
}
```

The supervisor body ensures that whichever parameter thread succeeds first is exposed and returned as the result of the whole concurrent computation. Since the other thread is neither exposed nor returned, its effect is implicitly discarded. If both threads fail, then the supervisor explicitly propagates failure to its invoker with the fail command. When either thread fails or completes the main loop exits and we check for failure of a. If it has failed, then we return the result of b, and likewise in the case of b being complete. Otherwise, a has completed or b has failed, and we depend on the result of a, bearing in mind that if the supervisor blocks on a failed thread, it will itself fail. The bindings to the identifier res in the final pair of blocks for the conditional ensure that the result of the computation is available before exposure takes place, by forcing the supervisor body to block on thread result. Although the logic of the alt supervisor is arguably quite complex, it is written only once, possibly by programmers with more expertise than those who use it. Focus contains the alternate computation supervisor within its standard library.

File update and environments

As mentioned previously, Focus unifies the concept of update for files and locations by allowing type equivalence between the *file* and *loc(string)* types. Consider the following program fragment. In Focus, locations are explicitly typed, and are dereferenced with the keyword 'at'.

```
let f = file("/focus/file.txt")      //map text file to a string location bound to f
let l = loc("foo")                   //create a location and bind to l
f := at l                            //write to file f as if it were a string location
l := at l ++ "bar"                   //append to string location
```

f and l here are type equivalent, and can be passed to functions that require either *loc(string)* or *file* types, and can be updated in the same way. Although f and l in the example are

indistinguishable, assignments to f have a different underlying semantics in that they are mapped to the file system. However, since files are effectively locations as far as the language and its run time are concerned, the supervisor environment model captures the semantics of concurrent file update in the same way as location update. Only exposure to the top-level thread can cause physical update of the file system. Consistency is ensured for IO communication between threads at a lower level in the same way as for shadowed locations.

Retrial

Earlier in this chapter, we described three control operations available to supervisor bodies: activate, suspend, and expose. One further thread control operation, retry, causes the child environment of a thread to be reverted to the state at its last capture, and the thread activated at the entry point of its inception. Reverting the environment to the point of last capture reverts it either to the point of last exposure, or to the point of thread inception if no expose has taken place. Retrial may be applied to threads in any state. Since retrial is in effect akin to the inception of a new thread, there are no transitions from failed and done to active shown in the thread state diagram presented earlier in this chapter. The perceived effect of retrial is the undoing of computation performed by the thread and re-execution. However, only computation since the latest exposure is discarded. If a thread has been exposed more than once, some update may have propagated permanently to the parent environment. If this update must be undone, then that is the responsibility of a higher level supervisor.

We can use the retry operation to repeatedly execute computations that fail non-deterministically, as shown by the following example.

```
let ret = supervisor( computation:int, maxLatency:int ) is {
        while not done computation do
                if latency computation > maxLatency do retry computation
        expose computation
}
```

A supervisor may implement failure semantics for concurrent threads based not only on their observables, but also based on the final value that they compute. Thus, the supervisor can implement a form of *acceptance test* similar to that of recovery blocks (discussed in the next chapter). Consider the following example.

```
let ret = supervisor( computation:int, minVal:int → int ) is {
        while computation < minVal do retry computation
        expose computation
        computation
}
```

Here, the computation is repeatedly invoked until it returns an integer result greater than or equal to minVal. Any side effect that computation may have is not made visible until it returns a satisfactory result. The reference to computation in the while loop conditional is in a value context, whereas the reference in the loop body is in a thread context.

Nested supervisor invocation

The expressions that represent supervisor bodies and threads are arbitrary expressions, and so may themselves contain invocations of supervised computations. Supervisor invocations in this context follow the intuition of function call. A supervised thread invoking a supervisor 'becomes' that supervisor from the point of view of the higher level supervisor. That is, execution properties such as suspension and rate observed by the higher level supervisor are those of the *invoked supervisor body*. That is, a supervisor observing the activity of one of its parameter threads cannot obtain the observables of any child threads the parameter thread may have through invoking a supervisor. Although we could conceivably design a mechanism whereby the observables of child threads were available, we believe that only the immediate supervisor body can interpret the significance of the observable properties of the computations it supervises, so these are not available to concurrent computation at a higher level.

The situation here is analogous to exception handling; it is generally accepted that the propagation of exceptions through multiple levels of abstraction is inappropriate. As functions invoke functions, the conceptual distance from the exception raising point to the high level handling point is to great to be able to handle the exception effectively at the higher level, since the context of its raising is no longer understood. We revisit exception propagation in chapter 8. Our analogy comes from the fact if high level supervisors could monitor the observables of child threads (as opposed to parameter threads) then their observations are crossing boundaries of abstraction, and could not reasonably be expected to form a basis for effective failure interpretation, since the conceptual distance is too great and the context of invocation unknown. In our discussion of the service combinator algebra in chapter three, we show how limit statements at a high-level of abstraction can prevent slow things from happening at a much lower level of abstraction, *even if performance is deemed acceptable at*

that level. Our semantics for nested supervisor invocation are intended to prevent this kind of behaviour, based on the belief that only the immediate supervisor is capable of interpreting thread failure reliably, and on the desire to keep programs simple.

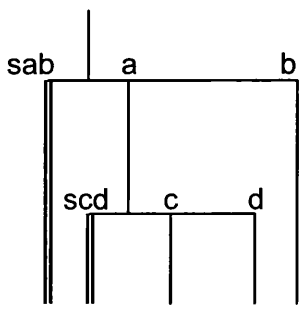
A major difference between nested function and supervisor invocations is in the propagation of failure. Failure is automatically propagated through dynamic function invocation chains. Similarly, failure of functions or supervisors invoked by a supervisor body is also propagated. However, propagation through a supervisor invocation chain stops when it reaches a supervised thread. This allows the supervisor body to determine whether to propagate failure or to attempt remedial action.

At any particular point in time, the current state of execution of a Focus program may be viewed as a tree, where nodes (forks, in the diagram below) represent supervisor invocations, and arcs (vertical lines) represent threads. Supervisor bodies are also threads at runtime, but we highlight their distinguished status with a double line. Every arc in the thread tree diagram has an associated environment. In the following example, the tree represents a point in execution shortly after the invocation of supervisor scd and before its return.

```
let sab = supervisor( a:int, b:int ) is {
        while ...
        let result = a        //block on a
        expose a
}

let scd = supervisor( c:int, b:int ) is {
        for ...
        expose c
}

sab( { let foo = scd(42,0); foo + 99 }, { ... } )
```



Here, the invocation of scd is nested within the thread a, which in turn is supervised by sab. The invocation of supervisor sab has no knowledge of the fact that an extended sub-tree exists beneath a. Queries of thread observables and state properties for a are dispatched to the supervisor body thread for scd. Thread control works similarly in that the thread being controlled is scd. Thus, sab cannot exercise direct control over c or d.The environment model described for single level supervisor invocation extends naturally to situations with

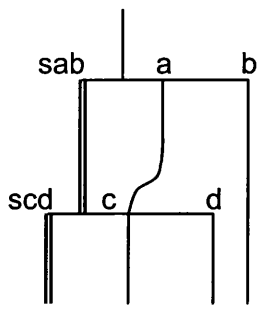
nested supervisors. Still considering the example above, the exposure of c by scd causes the unification of the environment of c with its parent environment, associated with a. Remember that the environment of a is distinct from that of sab, b, and the top-level parent environment. Updates performed by c and exposed by scd become visible at the top of the tree after the exposure of a by sab. It may help to think of exposure to the top-level parent environment as causing 'real' update to system store. However, the environment mechanism makes no explicit distinction between the top-level environment and other environments.

In addition to supervised threads invoking supervisors, supervisor bodies may invoke other supervisors. Consider the example below.

```
let scd = supervisor( c:int, b:int ) is { ... }

let sab = supervisor( a:int, b:int ) is {
        while ...
        let foo = scd(a,42)
        foo + 10
}

sab( { ... }, { ... } )
```



Here, it is the body of supervisor sab that invokes scd. In a sense, sab becomes scd, in a manner similar to that of a function application. Although sab is blocked on the return of scd, the threads a and b continue to execute. However, they cannot be controlled by sab, since it is inactive, synchronously waiting on the completion of the nested supervisor scd. This does not mean that a and b cannot be supervised, since they may be passed to any nested supervisor invocations as parameters. Since supervisor formal parameters are in a thread context, their passing as parameters to another supervisor does not block on their completion. Instead, the computations pass directly to the nested supervisor as threads. In the example, the thread a is passed to scd in this way. Thus, any thread operations invoked by scd on c are actually applied to the thread originally created and bound to a. Similarly, the observables of c are those of a. An important point that relates to threads passed between supervisors in this way is that threads always *retain the environment context of their inception.* This is significant for thread exposure. For example, if scd exposes the thread c, the environments unified as part of that operation are those of a and the top-level thread, as opposed to a and

sab. If the latter were the case, then separation of computation and control would be compromised, since side effect of a could be made visible in sab.

The previous two examples have served only to be instructive. To conclude this section we present two pragmatic examples of nested supervisors in action. First, we show how invoking a supervisor within an already supervised thread allows refinement and augmentation of failure semantics.

```
let limitRate = supervisor( foo:document → document ) is {
        waitfor rate foo < 0.25 and completion > 0
        if done foo then foo else fail
}

alt(    limitRate get "http://hostA.org/file.zip",
        get "http://hostB.org/file.zip" )
```

Here, the predefined supervisor alt is passed two computations one of which invokes a supervisor. The supervisor for alternate computation does not directly interpret failure according to rate, but by wrapping the computation we wish to limit within another supervisor that does, we can obtain the effect of limiting one branch of the alternate computation. The alt supervisor need not be modified or parameterised to this end.

The next example shows how we might split the specification of failure semantics into two modular supervisors: one for interpretation and one for recovery. We achieve this by invoking a supervisor from within the body of another supervisor.

```
let interpret = supervisor( foo:t, recover:supervisor( bar:any ) → t ) is {
        while not done foo do {
                waitfor time foo > 2.0 or done foo
                if not done foo do recover foo
        }
        foo
}

interpret(      someComputation(),
                supervisor( foo:any ) is
                        if completion foo = 0.0 then fail else retry foo
)
```

In this example, failure interpretation and recovery are isolated from each other. The interpret supervisor tries indefinitely to execute foo to completion. However, if it interprets failure by timeout, then it passes the still active thread to recover. The recovery supervisor knows how to recover foo from failure, but may decide that the failure is unrecoverable, in which case it propagates failure to interpret which, in turn, will fail. Note that the recovery supervisor does not even need to know the return type of foo.

This is a simple example, but real situations may demand sophisticated recovery or control policies for highly concurrent computations. Supervisors allow program aspects such as control, recovery, and failure interpretation to be modularly separated if required.

Examples of failure semantics

In this section we present supervisors that implement failure semantics similar to that which might be specified by a human browser. The first few examples implement logic equivalent to the human thought processes identified in the previous chapter, and so concern themselves only with failure interpretation. Later examples demonstrate sophisticated control, and we have already presented several earlier in this chapter. Finally, we present a small exercise in comparative programming between Focus and a Java-like language, in an attempt to demonstrate that Focus is more concise and intuitive when implementing in our intended application domain.

Note that most of the example supervisors contain literal constraint values. Given that the constraints are specified independent of absolute units of measurement, use of literal values in this way is perhaps acceptable. In any case, the constraint values could be passed as

parameters to the supervisor if necessary, but for the sake of brevity we do not. From this point, we return to the convention of emphasising keywords in bold text.

Human: "Don't waste time waiting for connection to a site that is probably down":

```
supervisor( computation:t, probThreshold:float → t) is {
        if prob computation < probThreshold then {
                waitfor latency computation > 1.2 or not active computation
                if done computation then computation else fail
        } else computation
}
```

This supervisor examines its probability of success of its parameter computation, and imposes a latency constraint if it is less than a specified threshold. Otherwise, the Web fetch is allowed to proceed unconstrained, since we expect success. In either case, success of the supervisor is dependent upon the success of computation. Since the result of computation is not relevant to the supervisor, we allow it to be polymorphic by specifying only that the supervisor return type (t) must be the same as computation.

Human: "Accept intermittent transfer from heavily loaded servers"

```
supervisor( computation:t, latencyThreshold:float → t) is {
        if latencyThreshold < latency computation then {
                let fails = loc(5)
                while active computation and at fails != 0 do {
                        if rate computation < 0.3 then fails := at fails – 1
                        else fails := 5
                        sleep 0.5       //allows 0.5*5 = 2.5 second rate trough
        }
        } else while rate computation > 0.3 do sleep 0.5
        if done computation then computation else fail
}
```

Before enforcing any constraints on the Web fetch this supervisor first checks to see whether the latency of the fetch was greater than a given threshold. If the fetch fails to connect, then this check will never be reached. We ignore this possibility for the purpose of brevity. If latency is higher than the threshold, we assume that the server is heavily loaded, and allow up to five successive rate constraint violations before interpreting failure. This approximates to allowing a two and a half second trough in the rate observable. Otherwise, we impose a simple rate constraint that causes failure on first violation. In Chapter 2 – *Analysing Web Failure and Performance*, we showed that under heavy network and server load, dynamic transfer rate is more variable and prone to troughs. Such a policy might be useful under these circumstances.

Human: "Invest more time in transfers that are close to completion":

```
supervisor( computation:t, timeConstraint:float → t) is {
        let constraint = loc(timeConstraint)
        while active computation do {
            if time computation > at constraint then
                if  completion computation > 0.8 and
                    timeConstraint = at constraint do
                      constraint := at constraint * 1.5
            else fail
        computation
}
```

In this example, the supervisor constrains the parameter computation by time alone. However, if the time constraint is violated, it checks whether the computation is close to completion before interpreting failure, and if it is the time constraint is loosened slightly. We compare constraint against the original constraint parameter to ensure that we grant time extension only once. Due to the notion of acceptability based on the norm, it can execute its failure interpretation over the full duration of computation, since the time constraint parameter will certainly be greater than the time observable of 1.0 for deterministic computation.

Human: "Be pessimistic in the morning, expecting downloads to take longer":

```
supervisor( computation:t, timeConstraint:float, highLoad:range of int → t )
 is {
        let constraint = if hourOfDay() in highLoad then
              timeConstraint * 2.0 else timeConstraint
        waitfor time computation > constraint or not active computation
        if done computation then computation else fail
}
```

Again, this supervisor constrains computation by time alone. However, it examines the time of day, and if within a certain period extends the given time constraint by a multiplicative factor. Similar to the last example, the supervisor can execute over the full duration of computation, since it can be fairly assumed that the given time constraint will not be violated by deterministic computations. That is, we assume that any actual parameter for timeConstraint will be greater than 1.0. This supervisor makes no assumption about the patterns in observable behaviour for completion. The computation may consist of any number of Web fetches, each of which is constrained in the same way. The time constraint parameter does not apply to the duration of the entire computation, but to the individual operations within it.

Human: "Expect high latency fetches to take longer overall":

```
supervisor( computation:t, timeConstraint:float → t ) is {
        waitfor rate computation != 1.0              //end of deterministic part
        waitfor completion computation != 0.0        //end of latency part
        let latencyTime = time computation           //store latency wrt time
        let constraint = if latency computation < 1.0 then timeConstraint
        else timeConstraint + (latencyTime/latency computation)
        waitfor time computation > constraint or not active computation
        if done computation then computation else fail
 }
```

This supervisor waits for the deterministic and latency portions of a computation to complete, then examines the latency that occurred. If the latency is better than expected (less

than 1.0), a given time constraint is enforced. Otherwise, the given time constraint is extended by the exact amount of latency overrun. Note that unlike previous examples, this supervisor makes the assumption that its parameter thread executes only a single web fetch, and synchronizes itself to the download portion of that fetch. However, it could be generalised to accept a computation that performs any number of web fetches by adding a while loop. However, we omit this for brevity.

Overall, these examples demonstrate that Focus can express the type of failure interpretation exhibited by humans in a concise and easily understood manner. Now we present some examples for which the emphasis is not only on failure interpretation, but also on control and recovery. In describing supervisors throughout this chapter, we have already presented examples of supervisors with various control flow characteristics. These include supervisors for alternate, priority, and parallel independent computation, as well as supervisors that use retrial to achieve behaviour similar to exception handlers and recovery blocks. We have also shown how nested supervisors allow failure interpretation and recovery policy to be modularly separated.

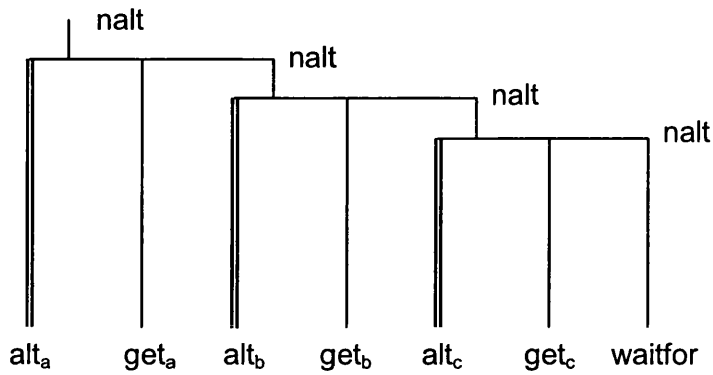
An earlier example presented the implementation of a combinator for binary alternate computation. If alternate computation of greater concurrency is required, additional alternate combinators can be implemented that take the required number of parameter computations. However, a more flexible solution is to allow alternate computation with a dynamically determined number of parameters. That is, we wish to approximate dynamic process creation with alternate semantics. The example below achieves this with a recursive function that invokes the binary alternate combinator.

```
forward nalt is function( set( string ) → document )

let nalt = function( urls: set( string ) → document ) is {
        if card urls = 0 waitfor false          //empty set termination case
        else alt( get take urls, nalt( urls ) )

nalt( "http://hostA.org/", "http://hostB.org/", "http://hostC.org/" )
```

The first line of this program fragment is a forward declaration that states the type of the identifier nalt (n-arry alternate). This is required because in Focus, identifiers are not in scope within their initialising expression, and so the type of the recursive call would otherwise be unknown. nalt recurses over a set of URLs, and the recursion is terminated by a waitfor that will never succeed. Eventually, one of the alternates will succeed or all will fail, and the recursion will unwind. Unlike traditional recursion, this 'concurrent recursion' can unwind from any point in the activation chain. For example, consider the thread tree diagram below corresponding to the program fragment.



If the fetch of a is the first to succeed, then the rest of the thread tree is immediately discarded. The example here is simple, with URLs being passed to nalt which in turn invokes a Web fetch for each. We can write a nalt function for arbitrary computation by wrapping up the computations in anonymous functions:

```
forward nalt is function( set( function( void → t ) ) → t )

let nalt = function( computations: set( function( void → t ) ) → t ) is {
        if card computations = 0 waitfor false
        else alt( take ( computations )(), nalt( computations ) )

nalt( set(      function( void → document ) is get "http://hostA.org/",
                function( void → document ) is get "http://hostB.org/",
                function( void → document ) is get "http://hostC.org/" )
```

This shows that we can approximate dynamic process creation using recursive supervisors.

Now we consider a more pragmatic example intended to optimise concurrent transfers over low bandwidth connections such as modems. Our experience indicates that with modem

connections, overhead in data transfer results in each of n concurrent transfers receiving significantly less than one n^{th} of the available bandwidth. We suspect that this is a result of overhead in multiplexing over the line. The following supervisor implements mutual exclusion of data transfers for concurrent computations that contain an arbitrary number of Web fetches. The intention is to explicitly timeshare bandwidth in a coarse grain manner to improve overall data throughput. The supervisor enforces mutual exclusion only in the case where both computations are attempting to stream data over the network, and does not restrict concurrency during connection phases or during deterministic computation. For brevity, the example handles only two computations.

```
let connectionMutex = supervisor( a:void, b: void ) is {

    let transferring = supervisor( c: void ) is
        completion c > 0.0 and completion c < 1.0

    let collision = loc(false)

    while not done a and not done b do {
        if failed a or failed b do fail
        if at collision then {
            if active a then {
                if not transferring(a) then collision := false
                else { suspend a; activate b }
            } else {
                if not transferring(b) then collision := false
                else { suspend b; activate a }
            }
        }
        if transferring(a) and transferring(b) then collision := true
        else { activate a; activate b }
        sleep 1.0
    }
    expose a
    expose b
}
```

The initial supervisor declaration by connectionMutex is a simple support supervisor that returns true or false based on an interpretation of whether the parameter computation c is transferring data across the Web. The collision location contains true if both computations are attempting to transfer data simultaneously. The main loop of connectionMutex continues until both computations complete, but causes overall failure if either computation fails. If the supervisor detects that both computations are transferring, it sets collision to true, and the mutual exclusion portion of the main loop is activated. This ensures that only one of the two computations is active, and checks whether it has finished transferring. If it has, then collision is set to false and both computations are activated. The sleep of one second at the end of the loop corresponds to the time slot given to computations in contention for bandwidth.

Our final example is one of comparative programming between Focus and a Java-like language. We use pseudo-code instead of pure Java for this so that the program logic is clear even to those unfamiliar with Java, and in particular its lightweight threading facilities. However, our pseudo-code accurately reflects the capabilities of the Java language model. Our task is to iterate over a set of URLs[1], checking whether or not they are available, and if not adding them to a set of bad URLs. This is a simple task, but we attempt a solution involving concurrency. The aim is to try and represent a class of applications that require integration of Web fetch, concurrency, failure interpretation, and failure recovery.

```
let urls = set( "http://...", ...)    //set of URLs
let badURLs = set()                   //accumulated set of failed URLs
let url = take(urls)                  //URL we are checking

let checkLinks = supervisor ( fetcher : void, updater : void )
{
  while not done updater do {         //keep going until no more urls
        while active fetcher do
              if latency fetcher > 2.0 or rate fetcher < 0.1 do {
                    expose fetcher    //commit to bad url
                    suspend fetcher   //will break loop
              }
        waitfor not active updater    //block until new url ready
        expose updater                //reveal update of new url to url loc
        activate updater
        retry fetcher                 //recaptures env, including new url
  }
}

checkLinks( { insert(url, badURLs); get url },                    //fetcher
  { while not empty(urls) do { url := take(urls); suspend } } )   //updater
```

The fetcher computation fetches the URL in the url variable, but pessimistically asserts it as a bad URL before doing so. The fetcher computation will only be exposed if it fails, so if it succeeds, the URL will not be recorded as bad. Concurrent with execution of fetcher,

[1] Note that in Focus, sets are not mutable, but for brevity and clarity of example here, we assume they are.

updater removes the first element from the set of URLs, and updates url with the next URL to be fetched. This does not interfere with the fetchers view of url, since the update by updater is not visible. The updater computation suspends itself in order to allow the supervisor to synchronise the overall computation. The supervisor repeatedly tries to interpret failure for fetcher, exposing the bad URL if it does, and synchronises re-execution of fetcher with the loop of updater. Now consider an implementation of similar intent with Java:

```
class URLContext { ... }        //persistence for observables, probably in library

class Fetcher extends Thread
{
  //We use Float objects rather than basic float so we can lock on access
  public Float latency;
  public Float rate;
  public URL url;

  public Fetcher(URL _url) {
        url = _url;
        latency = new Float(0);
        rate = new Float(MAXFLOAT);
  }

  public void run() {
        ... //open socket to URL, while locking and updating latency
        ... //get HTTP header and check return code
        while(socket.read(buffer)>0) {
              ... //calculate dynamic rate
              //we need a critical section for access to the observables
              lock rate { rate = dynamicRate; }      //locked for block duration
        }
        ... //update URLContext with new latancy and average rate
  }
}

class CheckLinks
{
```

```
    static Set urls = ...;
    static Set badURLs = ...;

    main(String [ ] args) {
        while(!empty(urls)) {
            URL url = (URL)urls.take();
            float avgRate = URLContext.getRate(url);    //persistent rate
            float avgLatency = URLContext.getLatency(url);
            Thread fetcher = new Fetcher(url);
            fetcher.start();                    //explicitly start a new thread
            while (fetcher.active()) {
                float latency, rate;
                lock fetcher.latency {
                    latency = fetcher.latency.getVal(); }
                lock fetcher.rate {
                    rate = fetcher.latency; }
                if (fetcher.latency > 2.0 * avgRate ||
                   fetcher.rate < 0.1 * avgLatency) {
                    badURLs.insert(url);
                    fetcher.stop();
                }
            }
        }
    //result is badURLs
    }
}
```

In the Java implementation, the CheckLinks class does the job of both the supervisor and updater computation in the Focus implementation. The observation and control aspect of the Java implementation is tightly coupled to the set operations. We can implement this task with separate threads for updater and fetcher, but this is complex since Java is not designed for implementing the observation and control paradigm.

We must ensure reads and updates to the latency and rate observables have mutual exclusion. This must be done with two separate locks. Consider the naïve alternative:

```
lock fetcher.latency {
        lock fetcher.rate {
                latency = fetcher.latency.getVal();
                rate = fetcher.latency;
        }
}
```

This could potentially result in deadlock, depending on the order in which locks are attempted by the fetcher thread. The implementation of locking here has been affected by the internal logic of the fetcher thread. Contrast this with the updater thread and fetcher thread in the Focus implementation.

Implementing these kind of concurrent programs in Java is more complex than with Focus, because it is difficult to model generalised control in Java. In Java, a controlling thread must have details of the internal logic and structure of the threads it controls in order to ensure mutual exclusion for concurrent updates. In summary, implementing this kind of task in Java results in the following:

- Tight coupling between thread logic.
- Lack of linguistic separation of control and computational logic.
- Computation specific control (control is only applicable to one computation).
- Requirement for explicit handling of shared variable update and rollback, in both computations and their controllers.

Supervisors provide a more concise and intuitive basis for implementing these tasks than Java, where there is linguistic and semantic separation of computation and control, generalised control, and support for concurrent update and automated backwards error recovery.

Summary

The Focus language contains an embedding of the supervisor and observables mechanisms. The supervisor construct is an abstraction layered over the persistent relative observables mechanism. Supervisors monitor the behaviour of concurrent computations and control their scheduling in order to drive failure semantics. Monitoring is effected by a distinguished thread, which appeals to the observables mechanism in order to interpret conditions of failure or synchronisation in a set of concurrent computations. A set of control operations allows the

distinguished thread to direct flow of control. Supervisors associate environments with individual threads, ensuring the separation of computation in the supervised threads and control in the distinguished thread. The environment mechanism provides automated backward error recovery and allows supervisors to explicitly control the communication aspect of cooperating concurrent computations. The supervisor mechanism has the following desirable properties:

- *Flexibility and orthogonality* – supervisors are first class entities, and there are no arbitrary restrictions on their use. For example, supervisors can be invoked from other supervisors or supervised threads, can be passed to functions or other supervisors, and can even form the result of function or supervisor invocations.
- *Highly concurrent* – supervisors are essentially concurrency constructors. Although the arrity of concurrency is specified statically in the declaration of each supervisor, dynamic process (thread) creation can be approximated using recursion.
- *Separation of computation and control* – the supervisor environment mechanism and type system with respect to threads ensures that computation in the supervisor body cannot interfere with computation in parameter threads other than in a controlling capacity.
- *Automated backward error recovery* – the supervisor environment mechanism means that all computation is 'hermetic'. Computational effect is committed only at the whim of the immediate supervisor for that computation. The decision not to expose the results of a computation is equivalent to discarding or undoing its effect.

In Chapter 3 – *Domain Properties and Flow Control*, we stated three criteria by which we evaluate Web programming systems. As they pertain to Focus and supervisors, these are:

- *Exposing the properties of the Web domain* – are the domain concepts orthogonal in that they compose in sensible ways with the rest of the language?

The domain concepts in Focus are language-level atomic Web fetch, which is an ordinary operation, and persistent relative observables. Since persistent relative observables are defined for all computation, the exposure of the domain is uniform across the entire language. The primitives that access observables can be applied to any computation. Thus, the exposure of the domain is an orthogonal concept.

- *What constitutes failure* – how flexible and orthogonal is the mechanism for failure interpretation?

In a supervisor, failure interpretation is flexible in that can be specified with arbitrary program logic, possibly involving the use of observables. Failure interpretation is independent of any particular computation, and is linguistically separated from program logic, implying orthogonality.

- *Flow control after failure* – does the flow control mechanism integrate with the rest of the language and in particular its concurrency mechanism?

In programming languages with exception handling mechanisms, problems arise when exceptions reach the locus of concurrency. This is because traditional exception handling mechanisms are serialised models. In contrast, the failure model for supervisors shifts the emphasis from detecting failure conditions inside a computation and propagating the information, to detecting them at a higher level, already in the context that knows how to handle them. This context is the body of the supervisor, which is also responsible for controlling concurrency. Consequently, programming logic responsible for detecting failure and driving concurrent flow control for failure is co-located, and intrinsically linked. In the traditional sense, supervisors have only a rudimentary exception mechanism. However, the flexibility provided by supervisors in unifying program logic for failure interpretation, concurrent flow control, and traditional flow control subsumes the functionality of sophisticated exception handling mechanisms. Supervisors and observables provide domain exposure, flexible failure interpretation, and flow control for failure all in a highly integrated model.

In Chapter 9 – *Formal Issues*, we present a correctness proof for an algorithm that allows for efficient implementation of the supervisor environment model.

7: Analysis of Related Work

In this chapter we examine work that is potentially applicable in the context of Web computation, or has similarities to aspects of our own research.

Recovery Blocks

Recovery blocks attempt to abstract over unreliable computations with software redundancy [68][69] and backward error recovery [70]. The mechanism provides functionality for the detection of failure in the form of an *acceptance test*, which is a boolean block expression that is linguistically separated from computational program logic. The acceptance test is evaluated on completion of a recovery block, and must be true for that block to complete successfully. On failure of the acceptance test, the mechanism restores the system to the state just before entry to the recovery block, under the assumption it is a consistent state, from before the manifestation of the error that caused failure. It then transfers control to the next alternate recovery block in sequence. On acceptable completion of an alternate, the overall result of the recovery block is as if none of the other alternates had been executed at all.

The recovery block mechanism is a general solution to the problem of when and how to switch to redundant code. It deals with both the restoration of state updated by the failing block, and with transfer of control to the redundant code. The recovery block mechanism does not attempt to diagnose the particular fault that causes an error, or to assess the extent of any other damage the fault may have caused. Instead, recovery actions return the system to a state prior to that of the introduction of the error then execute an alternate algorithm.

Failure of the acceptance test for the final alternate results in failure of the entire recovery module, and failure is propagated to the enclosing recovery block, if one exists. Failure of the entire recovery module also takes place if an error occurs during the execution of the acceptance test itself. The acceptance test cannot access identifiers declared locally to any of the alternate blocks, and so is at the static scope level of the enclosing block. However, to aid in establishing the correctness of a computation, acceptance tests have the useful property of being able to access the free variables of a recovery block not only for their current value, but also for their *original* value before entry to the recovery block. Each alternate attempts to pass the same acceptance test, but they do not need to produce the same results. This allows the programming methodology whereby alternates to the primary computation provide only a degraded service; or no service at all, simply relying on the mechanism to recover a consistent state.

An acceptance test is similar to the post-condition of a procedure's formal specification. However, formal specification languages are usually at a much higher level than the language they specify, often containing quantifiers, for example. This is problematical, since in program logic it can be just as complex and error-prone to evaluate a post-condition as it is to compute the result being tested. Thus, for pragmatic reasons it is usually necessary to adopt a somewhat less effective acceptance test than that specified by a program specification. The

choice of an appropriate acceptance test must be a trade-off made by the programmer according to the required levels of robustness and efficiency.

There may be occasions when it would be convenient for the alternate to know the circumstances of failure in a previously terminated alternate. However, the number of possible error conditions may be very large, and it is not always easy to distinguish one error from another. Melliar-Smith argues [105] that it is unreasonable in many cases to expect an alternate to categorise and accommodate each possible failure explicitly. This is supported by Randell who claims that,

> "...errors which are expected to be sufficiently frequent that special handling would be appropriate can perhaps be regarded as normal program conditions rather than unforeseeable errors." [71]

Therefore, alternate blocks are defined as being independent and transactional in that they either execute to completion, pass the acceptance test and form the result of the whole recovery block, or the system reverts to the state before the execution of the alternate began, and executes the next alternate.

Although alternate blocks are all physically independent from each other they need not be logically independent. For example, it may be efficient to use a fast heuristic algorithm that 'almost always works', and when an exceptional case is discovered, use a slower algorithm that 'really always works' instead. In this example, the result of one alternate block is more desirable than another, less efficient block. In general, a set of alternates encompassing all acceptable behaviours can be designed and a preferred sequence specified by using the linear ordering of alternate blocks in program source. This allows a definite structuring for error recovery facilities.

The recovery block mechanism is orthogonal in that it can be applied to almost any language, even those as low-level as assembler. The only requirements are that the recovery blocks should be explicitly defined, that they should be dynamically nested (associated with block or procedure activations), and that entry and exit from recovery blocks should be explicit.

Backward error recovery is difficult to apply in the structuring of concurrent systems that achieve 'progressive' computation in the face of errors. In systems that consist of communicating processes, state restoration in one process forces cascading state restoration for all processes dependent on the state being reverted. According to Randell, this frequently

leads to a 'domino effect', where large amounts of computation are abandoned and restarted. In any case, for systems that communicate with the external world, state restoration may be impossible, even in principle. This is known as the "please ignore incoming missile problem".

A solution to this problem was proposed with *coordinated* (or *concurrent) atomic actions* [66], which constrain the setting of recovery points in concurrent computations. Concurrent atomic actions force entry to recovery blocks to be coordinated with all peer processes. This provides a 'fixed-point' for rollback that avoids cascading failure. The concurrent atomic action is the state of the art in coordinated error recovery, but is a heavyweight solution that is primarily intended for distributed and process concurrent systems.

Supervisors employ a mechanism for automated backward error recovery that is broadly similar to that of recovery blocks in that state restoration is implicit. In contrast to the processes of concurrent atomic actions, supervisor threads are lightweight. We feel that the adoption of such a mechanism would add unnecessary complexity to the supervisor construct. If distributed failure must be modelled in Focus, applicable language independent approaches have been identified that implement distributed concurrent atomic actions [61].

Recovery blocks were originally designed to achieve *software fault tolerance*, which attempts to automate recovery on manifestation of unanticipated programmer design flaws or logic errors in program source. In addition to programming errors, Randell argues that they are also useful in masking anticipated exceptional errors [72]. However, in this context recovery blocks are less flexible than supervisors. Foremost in this is the fact that recovery blocks have a fixed flow of control on the detection of failure. The acceptance test is a programmed failure detector for a computation, but it cannot direct control flow dynamically, since the selection of which alternate algorithm to execute specified in the static ordering of alternate blocks. An example of why this is inflexible is the situation where the acceptance test is capable of dynamically determining from the circumstances of failure that execution of a particular alternate is likely to succeed. Despite this knowledge, the alternate invoked is the next in the statically specified sequence.

Perhaps the most important difference between supervisors and recovery blocks is that unlike supervisors, recovery blocks cannot specify constraints on the dynamic observables of a computation, since the acceptance test is evaluated only as a post-condition. Failure can only be interpreted *post-hoc*, by analysis of update to the system's global state. Thus, the recovery block mechanism provides no support for the interpretation of failure during the execution of non-deterministic operations. Primarily, this is because recovery blocks were originally designed to abstract over software faults.

Supervisors can implement the recovery block mechanism, while allowing a more general solution to providing software redundancy. The solution involves use of an acceptance test thread in conjunction with a supervisor, and so is perhaps more complex than the simple recovery block mechanism. However, if necessary:

- supervisors can retry failed alternates,
- alternates can be executed concurrently and independently for efficiency,
- alternate ordering sequence can be determined dynamically,
- supervisors are parameterisable by their redundant computations,
- the 'acceptance test' can retain state between alternate invocations, and
- the test is not a post-condition, but can detect failure dynamically as it occurs.

To summarise, the main benefit of recovery blocks is that the programmer need not be concerned with enumerating all possible failures, and can rely on the automated backward error recovery to restore the system to a consistent state. However, recovery blocks are less flexible than supervisors in that they have fixed flow of control on detection of failure – execution of the next alternate in sequence. In addition, acceptance tests provide no support for the interpretation of failure *during* the execution of the block, since they are essentially *post-hoc* conditionals. Supervisors can wholly implement the recovery block mechanism, and provide greater flexibility in both the interpretation of failure and the specification of control flow.

Concurrent Transaction Control Techniques

According to FOLDOC [73], a *transaction* is "a unit of interaction with a database or similar system that must be treated in a coherent and reliable way independent of other transactions". That is, a transaction must be logically atomic, even though it may be composed of several distinct interactions. The purpose of transactions is to prevent inconsistencies arising through concurrent access to shared resources. The classic example of this is where two distinct processes in a banking system try to perform simultaneous updates to an account. Each update requires several distinct operations. If there is no transaction mechanism to protect the system, then the overall result can be one of several (incorrect) states, depending on how execution of the two different processes are interleaved. In order to protect system integrity, concurrency control techniques must be applied so that transactions are *serializable*. That is, they are

atomic and logically ordered in their effect, despite the fact that they may be executed concurrently.

Concurrency control schemes for transactions are generally categorised into two areas: pessimistic and optimistic. *Pessimistic* schemes are based on the assumption that transactions frequently access and modify small 'hot spots' of data, and thus commonly interfere with each other when run concurrently. These schemes force processes to obtain a lock on the data that the transaction is accessing before it begins. If another transaction has locked the data, then it cannot proceed, and must retry later. Two phase locking [74] is the most widely known pessimistic concurrency control technique. It is designed to avoid the possibility of process deadlock, but the 'back-off and retry' strategy it is based on can lead to process starvation in some systems. More significantly for us, is that pessimistic schemes rely on the programmer to perform the locking.

Optimistic schemes are based on the assumption that transactions rarely access the same data structures, and so there will be few access conflicts between transactions. Given this, optimistic schemes speculatively allow transactions to proceed, and when they are ready to *commit* their effect to store, check to see whether any conflicts did actually occur. If there was a conflict, then the effect of one transaction is rolled back to a state before it began. After this, the transaction can be retried a later time. Most modern database systems, such as Oracle [75], for example, provide optimistic transactions.

The supervisor environment mechanism is not a transaction mechanism, but it does have similarities to optimistic transactions. The main focus of supervisors and the environment mechanism is to take the responsibility of concurrency control out of the hands of the computations themselves and make it the primary obligation of the supervisor construct. Similarly, optimistic mechanisms do not require the transaction computations to deal with concurrency issues.

Transactions are intended to be 'serializable', meaning that concurrent transactions behave as if they were executed sequentially, even though they may actually be interleaved. Thoughout their execution, transactions see a 'snapshot' of the store as of their start time. This is similar to supervised threads. However, the commit (expose) of supervised threads compromises serializability. Updates by one thread can be 'merged' with another in a manner dependent upon the order of exposure, and no facility is provided to ensure serializable computation. However, environment exposure is intended primarily as a means to allow rollback of computation (by not exposing) and to allow communication between threads, and not to commit transactions.

Transactions allow operations on objects to be grouped together and provide the atomicity guarantee – all or nothing execution. Focus environments provide the same functionality, as it is important for rollback after failure. However, optimistic transaction mechanisms provide automated support to recognize when data is potentially compromised by certain concurrent interleaving computations. For the class of applications for which Focus is intended we do not believe that this is enough of an issue to provide this kind of support directly. However, a future line of research could be to integrate such a mechanism if these semantics are found to be desirable.

LogicWeb

The language *LogicWeb* [76][77] is an extension to the logic programming language Prolog [78]. It permits programmable behaviour and state to be associated with Web documents, allowing them to be queried using knowledge-base logic program representations. LogicWeb allows the programmer to think of Web computations as goals applied to programs, with no need for explicit Web page retrieval or parsing. It is a high-level model built around the notion of structured data, distributed across the global network, and coupled by logical relationships. A program incorporated into a page can reason with other Web documents as part of its behaviour.

LogicWeb is a 'mobile code' system in that program representations are transmitted across the network and executed at a destination site. LogicWeb code physically moves from the server host to the client host in order to execute. This local execution model is analogous to that of Java applets and JavaScript programs, which are logically embedded within an html document and migrate with it to be executed at the local host. The design of LogicWeb is geared to the client-side evaluation of logic goals by the manipulation of multiple programs from disparate sources.

A LogicWeb *module*, which corresponds to an HTML document, contains a program written in a version of Prolog extended with new operators. These operators allow the retrieval of other LogicWeb modules, and the invocation of goals within them. The Web itself is viewed as a directed cyclic graph of program modules. When modules are downloaded, their predicates are installed into the current environment, and goal evaluation continues. In essence then, LogicWeb allows the components of a logic program to be distributed across the Web, and integrated on demand.

LogicWeb is an interactive system, and requires the compiler to interface with a browser application. LogicWeb appends an HTML form interface to every page viewed through the browser. This allows the user to interact with the system by entering and submitting a goal, which is redirected to Prolog for evaluation. LogicWeb also responds to input with the mouse.

Each time a hyperlink is traversed by a user, the corresponding document is fetched by the browser, and installed into Prolog as a new module. By default, all Web documents install the facts *my_id*, *h_text*, and zero or more *link* facts. These correspond to the document URL, HTML source, and the URLs embedded within the document. Thus, all Web documents can be viewed as LogicWeb modules. However, some Web documents may contain specially marked up Prolog code. Facts included in this code are installed into the current environment, and may specify arbitrary information. Most interesting, though, is that relationships between modules can be expressed by predicates involving the URL link facts, allowing pages to specify the meaning of their hyperlinks. This code can be declarative in that it specifies structural relationships, or can be code that performs some function by executing on link activation, or both. After the Prolog code, if any, has been installed, the browser displays the HTML of the *h_text* fact.

There is no persistence of information across browser sessions, as all installed modules are lost on ending a session. Moreover, when evaluating a goal in a remote module, the module is only downloaded once. Further invocations of that goal will use the same module. This is to enforce consistency in query results between goal evaluations. Otherwise, a program may fail where it previously succeeded or succeed with different bindings, behaviour that is problematical in a logic programming system. Web pages tend to change infrequently, and so LogicWeb assumes all pages to be constant over the duration of a computation. Failure to download a particular document required for goal evaluation results in failure of the entire goal. Such failure is not directly distinguishable from normal failure of a goal. However, an ad hoc measure is to evaluate the 'true' goal in the context of a remote document, in which case it fails if that document cannot be retrieved. Subgoals that fail because of download failure do not cause other modules downloaded as part of the overall goal to be retracted. This means that there can be a side effect from the failed goal. However, it is possible to manually keep track of the modules that are loaded and retract them on goal failure if necessary.

The LogicWeb view of the Web as a distributed collection of program modules hides the pragmatic concerns of network latency, bandwidth, and non-determinism. With the current implementation, failure semantics involving timeout or rate limit cannot be specified. However, recent work by Davidson describes possible additions to LogicWeb whereby these concerns may be addressed [79]. The proposed augmentation allows communication between client and server to be viewed as a stream of data passing between logic programming processes. This is possible by adapting LogicWeb for the relatively new Concurrent Logic Programming (CLP) paradigm [80]. CLP allows the viewing of programs as networks of processes connected by streams of data. With CLP, logic programs can implement abstractions that make use of stream AND- and OR- parallelism. These allow several kinds of

interactions between processes, including many-to-one, broadcast (one-to-many), and blackboard (many-to-many) communication. The proposed mechanism uses a producer-consumer and stream viewpoint in order to represent http responses. This allows Web concepts such as download failure, latency, time-out, retry, and transfer rate to be captured at the program level. In principle, the mechanism also provides the building blocks for specification of more complex behaviour.

The current implementation of LogicWeb extends the Mosaic Web browser. Mosaic supports the Common Client Interface (CCI) [81], which allows interaction with the browser application through socket streams. The limitations of CCI mean that a CGI script (typically on a local server) must receive each user query, and construct a canonical goal from it that is then returned to the browser. The browser redirects the canonical goal to Prolog for evaluation, and displays the result in the browser window. This indirection could have been made redundant if a solution involving a Prolog interpreter written as a Java applet or with JavaScript were adopted.

LogicWeb relies on integration with a browser application, since it was designed as a query language and tool for augmenting the browsing experience. The LogicWeb programming system and the Mosaic browser can be distributed independently, since the use of the CCI does not require modification of Mosaic at the source level. However, widespread use of LogicWeb is hampered by the fact that the popular Netscape and Internet Explorer browsers do not support the CCI. A solution involving Java or JavaScript would allow uptake of LogicWeb by anyone with a browser supporting that technology.

Unlike Focus, LogicWeb views the Web holistically, as a set of distributed programs. However, executions of LogicWeb programs are not distributed. This is because all computation takes place at the client side, and because the assumption of document consistency for the duration of a LogicWeb session prevents direct communication between LogicWeb agents. In general, integration with non-LogicWeb applications on the Web (such as search engines, for example) is not possible, since LogicWeb provides no support for the http post method. LogicWeb is primarily a language that allows the creation of interactive logic systems for the Web. However, as a language, it is significantly higher level than Focus, and its target application domain is correspondingly smaller.

The use of CLP and streams allows LogicWeb to implement failure semantics similar to that possible with supervisors. However, LogicWeb provides no historical context for URLs. It is possible, though, that these could be implemented reasonably concisely in an ad hoc manner. In any case, the proposed CLP extensions to LogicWeb have yet to be implemented.

Real-Time Languages

Real-time (RT) programming languages [82] are designed in part so that their programs can be checked for adherence to critical timing constraints. There is a domain overlap between RT languages and Web programming languages since time is intrinsic to the interpretation of failure in some Web applications. RT languages allow a degree of control over timing issues with constructs in the language semantics, and so the design decisions relating to these constructs are of interest to designers of Web programming languages. By contemporary criteria, it is important that the programs of a real-time language be *schedulability analysable*, where the maximum time of execution is known statically for all parts of the program [83]. It can then be statically determined whether or not the processes of a schedulability analysable program fulfil their timing constraints. Achieving schedulability analysis at compile time has major implications for language structure, since concepts that cannot be statically time-bounded must be elided, such as unbounded recursion and dynamic memory allocation, for example. However, analysis of schedulability is important primarily in the context of *hard real-time systems*. These are systems in which the failure of a computation to meet its deadline results in catastrophic (unacceptable) failure of the entire system, such as in flight or reactor control, for example. The failure-prone nature of the global network dictates that it is an inappropriate medium for the implementation of hard real-time systems, so in analysing the domain overlap between real-time programming languages and Web programming languages, we do not consider facilities for schedulability analysis.

The following three sections describe real-time programming languages for each of which some aspect of their design merits a comparison with Focus.

Process Control Language

The Process Control Language [84] (PCL), is perhaps the earliest real-time PL/I dialect, and defines a rich real-time tasking model. Although the dialect is somewhat dated (1969), fulfilling few of the criteria of a 'good' RT programming language that the current state-of-the-art demands [85], we found one of its design features of interest in the context of a comparison with Focus. PCL allows the declaration of variables to be augmented with the *analog* (sic) attribute, specifying that the identifier represents an external signal rather than a normal variable. The analogue attribute is designed to allow high-level interfacing with external hardware, and so may be specified as being readable, mutable, or both. Once an identifier is declared as analogue, attributes may provide additional information about it. These are described in the following list.

- *History* – analogue variables can be declared as having a *history*, of parameterisable length. The history is updated when the variable is input or output, and may be queried throughout the program. In addition, the history can optionally store the times at which the history values were input or output. All analogue variables have a history of one by default, allowing the programmer to query at least the previous value of an analogue variable.
- *Limit* – each time a signal is input or output its value is checked against specified upper and lower limits. If either limit is exceeded, an exception is raised.
- *Scale* – this allows the modification of the value on input or output by an arbitrary expression.
- *Access* – this specifies how the signal is to be read or written. The access attribute allows analogue variables to be orthogonal in that they are applicable in any context where a normal variable is valid, such are array construction, and on the left or right hand side of an assignment, for example. In particular, read operations on analogue variables may be overloaded as use of the analogue variable's identifier in normal expressions. The different values for the access attribute are as follows:

    - *Reference* – the analogue variable is input or output each time its identifier is encountered in an executable statement. Variables declared as output may only appear as l-values.
    - *Command* – the signal is input or output only when specified by an explicit command.
    - *Period* – the signal is input or output automatically, updating it with a given time period.
    - *Interrupt* – the signal is input or output each time an event specified by an (arbitrary) interrupt expression is true. The interrupt expression is given as a parameter to the access attribute.

Consider the following example, from [84]:

DECLARE ALPHA **ANALOG INPUT**
SCALE (2*ALPHA+A)
LIMITS (2.4, 8)
ACCESS (**REFERENCE**)
HISTORY (20, 4, **TIME**);

The variable ALPHA is declared as being analogue input only, meaning that the identifiers refers to some external device from which the program will read values. The variable is scaled according to an arbitrary expression, is limited with a lower and upper bound, can be used in an arbitrary expression but not as an l-value, and maintains a history of twenty previous values accumulated from every fourth input operation. In addition, the history stores the time at which the values were read.

Given a suitably augmented implementation of analogue variables, embedded within a programming language with a primitive Web fetch operation, the combination of these facilities provides a level of abstraction analogous to that of the persistent relative observables mechanism. The use of scale, history, and limit in combination allows the calculation of relative observables and specification of their constraints, with less syntactic overhead than implementing similar functionality in a language without analogue variables. However, such an implementation would be less concise than the Focus implementation of observables. This is because with analogue variables, observables must be explicitly implemented for each operation, the observables are distinct, and there can only be loose coupling between the observables and their associated operation.

FLEX

An intrinsic concept in FLEX [86][87] is that of imprecise computation. The FLEX language is geared to the writing of programs that incrementally produce results whose precision monotonically increases over time. The major novel contribution of FLEX is that it allows the dynamic substitution of a less time consuming computation for a time consuming one, affording programs a greater likelihood of meeting their time constraints.

FLEX supports a rich variety of timing and resource constraints, which may be both static and dynamic. In particular, it provides the constraint block, which is a language construct that allows the programmer to express relationships between variables. These must be enforced throughout the execution of a program block, and an exception handler may be provided for situations in which a constraint cannot be satisfied. Constraint blocks have several associated *attributes*, which are updated on the start and finish of execution. The most interesting of these are *start*: the absolute time that execution begins, *finish*: an absolute time, and *duration*:

a relative time less than or equal to the interval between start and finish. The values of these attributes are retained between executions of the constraint blocks.

In FLEX, constraints are boolean expressions that can be formed from both constraint block attributes and normal variables. Constraints are periodically and automatically evaluated by the run-time system and an exception is thrown if they are not true.

FLEX is an object-oriented language, and constraint blocks are overloaded as objects. Constraint block objects are named, and they can be referred to by their own code, or by the code of other constraint blocks. This allows concurrent processes to work together in maintaining their constraints and for constraint blocks to compute with the timing history of other constraint blocks. Consider the following example:

```
CB1: (duration < (5-CB2.duration)) and (duration<4) { ... }
CB2: (duration < (5-CB1.duration)) and (duration<4) { ... }
```

Here, two processes corresponding to constraint blocks collectively must finish within five minutes, and each process must take no longer than four minutes. For example, if CB1 has executed for three minutes, then CB2 must complete within two minutes. When CB2 examines the duration of CB1, CB1 need not be executing and can be complete, since constraint block attributes are retained after completion of the block.

The following code shows how a persistent time observable can be implemented for constraint blocks so that they can be specified in relative terms.

```
float CBDurationHistory := ...
integer histCount := ...

CB: duration < CBDurationHistory * 2.0 ~> timoutError() {
  ...
  CBDurationHistory := (((CBDurationHistory*histCount)+Clock.get_time-
                         CB.start)/(histCount+1)
  histCount := histCount + 1
}
```

This example specifies that the constraint block should take no longer than twice the time taken before for that constraint block, on average. Otherwise, the timoutError exception is thrown.

Real-time Euclid

RT-Euclid (RTE) [88] is modular, strongly and statically typed, and contains features that make it real-time and fault-tolerant. The RTE mechanism for device access provides language level features that could be useful in a Web programming context. The syntax is as follows.

```
var <id> device atLocation <intExpr> : <typeSpec>
        [noLongerThan <timeExpr> : <timeoutReason>]
```

When the identifier id is referenced as an l-value or r-value, the device is activated and a value is read or written, as appropriate. If the operation takes longer than timeExpr to complete, then timeoutReason is raised as an exception. With an appropriate augmented embedding (replacing the absolute memory address with a URL) in a language with a primitive Web fetch operation, this mechanism could map simply to a means of downloading Web documents. Use the identifier id would invoke a download of the document, which the run-time system would then attempt to convert to type typeSpec.

Summary

Recovery blocks provide the useful concept of automated backward error recovery, but can only detect failure as a post-condition. The onus is still on the computation itself to detect the conditions of dynamic failure.

LogicWeb attempts to apply the logic-programming paradigm to the Web, and does address some of the issues arising from the Web's failure and performance properties. However, the target application domain for LogicWeb is more limited than that of Focus.

Real-time languages are concerned primarily with timing constraints for computation. PCL, FLEX, and RTE all include language level concepts that are similar in some way to aspects of the supervisor mechanism. However, in each case supervisors and persistent relative observables provide more generalised functionality. Primarily, this is because real-time languages deal only with time, and the many additional observables directly available in Focus provide added flexibility.

8: Exception Handling

Exception handling mechanisms are the only widely adopted programming language abstractions related to failure. The languages C++, Java, and Ada95 account for much of contemporary software implementation, and all three incorporate an exception handling mechanism. Exception handling mechanisms can take many forms and in the next few sections we attempt to outline their taxonomy. The main goals of exception handling mechanisms can be summarised as follows:

- Reducing the number of explicit error tests.
- Automating flow control after detection of errors.
- Separating error handling code from computational logic.
- Bringing failure within the language model to replace ad hoc methodology.

Programmed exception handling involves the predicting of faults, such as hardware failure or corrupt input, and their consequences, such as whether the program can continue by taking special action. It is a mechanism designed to preserve structural clarity in a program by linguistically separating algorithms that deal with detected errors from those that may generate them, and by automating error propagation and error handler selection. The programmer is still responsible for detecting and indicating the presence of errors explicitly in program logic, but exception mechanisms eliminate the need to 'redetect' errors at every level of function activation. This reduces the number of explicit tests that are required.

Since there is a wide range of exception handling mechanisms, the associated terminology is diverse. We attempt to compromise as much as possible in this respect. We shall assume the terms *procedure*, *function*, and *abstraction layer* as having broadly the same meaning. The latter, however, has additional connotations of an explicit program *module,* possibly containing several procedures or functions. The *activation* of a procedure or function implies the crossing of a layer of modular abstraction. The *signaller* is the abstraction layer that *raises* (synonym *signals* and *throws*) an exception. The *invoker* is the abstraction layer that causes execution in the abstraction layer that ultimately raises an exception.

Goodenough originally defined *exceptional conditions* as "those brought to the attention of an operations invoker, which become part of the normal exit or return" [89]. However, this definition has been criticised as being too general [90][91][92]. Gehani [55] adopts a more specific definition of exceptions as "an error or an event that occurs unexpectedly or

infrequently". The most commonly accepted definition of exception, and the one assumed here, is the union of 'error', 'exceptional case', 'rare situation', and 'unusual event'. We do not include the notion of 'unexpected event', since to be captured by an exception handling mechanism, events must by the very nature of exception handling mechanisms be expected, however infrequently. A programming abstraction that can raise an *exception*, either explicitly or implicitly, may be *guarded* by an exception *handler*. On the occurrence of an exception, control is passed to the handler, which decides what action is to be taken. According to Goodenough, who presents the seminal work on exception handling,

> "...exceptions permit the user of an operation to extend [an] operation's domain – the set of inputs for which effects are defined – or its range – the effects obtained when certain inputs are processed." [93]

In other words, exceptions allow generalisation of the abstractions that may raise them, by defining their behaviour in cases that without exceptions would have resulted in error. Exceptions are the means to represent a particular kind of exceptional circumstance. They may be simple identifiers, special or ordinary data types, data structures, procedures, or messages.

The majority of exception handling mechanisms are based on Goodenough's proposals, and so define exceptions and handlers separately. Mechanisms that declare exceptions and their handlers together, or unify them, do so out of the desire for completely statically typed mechanisms. Knudsden's *sequel* construct, described later, is a mechanism that unifies exceptions and their handlers in this way.

Exception raising

Exceptions are primarily a vehicle for the propagation of error information from a lower level of abstraction at which it cannot be reasoned about to a higher level of abstraction that can determine its significance from the context of occurrence. Once an exception is raised, exception mechanisms can differ in the number of abstraction levels over which information is propagated automatically. Horning states that

> "...[an exception handler] can be placed at a level in the system where there is sufficient global information to effect a reasonable repair, report the problem in more user-oriented terms, or decide to start over. However, the

more levels [of abstraction] through which [an exception] passes before being handled, the greater the conceptual distance...between the signaller and the handler." [94]

Yemini [95] and Liskov [53] argue similarly, asserting that multilevel propagation increases coupling between the handler and signaller, compromising modular information hiding since details of the signaller's internal implementation are (required to be) exposed at higher and intermediate levels of abstraction. Their arguments contend that only the immediate invoker of an abstraction knows the full significance of any exceptions that are raised from it, and so automatic propagation of unchanged exceptions through abstraction layers should be precluded. If exceptions must be propagated through more than one level of abstraction, they must be explicitly raised again. This encourages programmers to re-express the exception in terms more meaningful at the higher-level of abstraction.

Yemini's mechanism, Levin's mechanism [53], the revised Algol68 [96], and Clu [97], all allow only a single level of exception propagation. Cristian [90] and Anderson [98] also support the concept of single level propagation. Goodenough's notation, C++, PL/I [99], Mesa [100], Ada [101], and WebL[1], among others, automatically propagate exceptions through any number of abstraction layers until they are handled.

The *exception interface* is the part of an abstraction's interface that explicitly specifies the exceptions that might be raised or propagated by that abstraction. This enables static consistency and reliability checks. Both Yemini's mechanism and Goodenough's notation require the static checking of all procedure interfaces in order to ensure handling of all possible raised exceptions. Although this eliminates a large number of potential programming errors, it can be a burden to the programmer, since the raising of certain exceptions may have been precluded by program logic. For example, a possible divide by zero may be obviated by a dynamic test in program logic, but both mechanisms still require the specification of a handler for just such an eventuality since the logic that precludes it cannot be detected mechanically. Knudsen's sequel construct is also entirely statically checked, but because of its unification of exceptions and handlers, it is different enough to merit discussion in a separate section (below). All other mechanisms use at least some dynamic checks.

A problem with any exception handling mechanism that does not consider potentially raised exceptions to be a part of an abstraction's interface is that it is possible for indirect (and even direct) exception propagation to be completely overlooked until it causes catastrophic failure.

---

[1] We present a detailed survey of the WebL exception mechanism in a later chapter.

This can be due to indolence on the part of the programmer or, as pointed out by Horning [102], the fact that non-static exception interfaces tend to be the least well documented and tested part of an abstraction.

Exception handlers that are declared at a different level of abstraction from the raising context of its exceptions cannot directly access data local to the signaller. To counter this, several mechanisms allow the generalisation of exceptions by parameterising them at the time of raising. This allows the handler to take more specific action in response to a particular exception context. However, the amount of information that is conveyed can be limited by the parameterisation mechanism. Mechanisms in which exceptions are objects have no restriction on the passing of information, since the exception can be of arbitrary type, and carry as much information as necessary. C++ [103] and Java [104] are examples of languages where exceptions are arbitrary objects.

Mechanisms such as that of Ada [101] do not allow the parameterisation of exceptions, limiting the conveyance of information to the act of raising the exception itself. If parameters are required in such mechanisms, one option is the use of global variables to temporarily hold state. However, such a methodology undermines the original purpose of an exception handling mechanism, and can be difficult to achieve correctly in the presence of concurrency. An alternative is to declare a separate exception and associated handler for each possible raising point. Each exception is uniquely tied to a particular raising context. For example, instead of declaring a single parameterisable file IO exception, the programmer declares many different file IO exceptions, one for each possible parameterisation. This increases complexity and decreases program strength since many different handlers with similar functionality must be developed.

Melliar-Smith points out [105] that the number of possible failure modes of a module increases rapidly as its internals become more complex. He argues that it is impractical to enumerate all the possible failure modes, let alone design algorithms to detect and handle all possible failures individually. This is a compelling argument for at least some form of exception parameterisation. Object exceptions provide the most flexible form of parameterisation. Exception handling mechanisms for languages with object exceptions and inclusion polymorphism (Java and C++) are even more flexible, since handlers can be defined for entire exception subclass hierarchies.

Handler response

With the *termination* model of exception handling, activation of the handler results in the immediate termination of all procedure or block frames that the exception propagates through.

In contrast, the *resumption* model allows the handler to resume the signaller after attempting remedial action, at the operation following the one that caused the exception.

The resumption model is most easily understood by viewing exception handlers as implicit procedure parameters to the signaller. The invoker of an operation can declare handler functions that are then passed to the operation as additional parameters. Internally, the operation invokes a handler function in order to 'raise' an 'exception', passing parameters from its own scope to the handler. Once the handler completes, presumably having taken remedial action, control is returned to the signaller and execution resumed at the operation following the handler invocation. If a language has higher-order procedures, the resumption model can be approximated without recourse to any explicit exception handling mechanism.

Both Goodenough and Levin [53] favour the resumption model of exception handling. They base their opinion on the argument that resumption can preserve valuable state information that may have been accumulated by computation taking place before the exception is raised. In contrast, Liskov argues [106] that the expressive power of the termination model is adequate with respect to the resumption model, in that programming situations resolved awkwardly with the termination model and simply with the resumption model are infrequent. Moreover, she contends that the resumption model results in unnecessary coupling between the signalling abstraction and the exception handler. Cristian [90] and Anderson [98] provide similar arguments favouring the termination model.

Assuming the absence of mutual recursion, abstraction layers are normally hierarchical in that their behaviour is dependent only upon the behaviour of any abstractions they themselves invoke as part of their execution, and not on the abstraction that originally invoked it. Moreover, hierarchical abstraction layers are modular – in order to understand the implementation of a procedure it should not be necessary to examine the implementations of the procedures it invokes. The termination model of exception handling retains these desirable properties, but the resumption model does not. With resumption, the caller and signaller are mutually dependent since in the event of an exception the signaller passes control back to the caller, which then must modify state in order to correct the execution of the signaller before returning control. The invoker must understand aspects of the signaller's internal logic in order to remedy the exception. This compromises modularity and the abstraction hierarchy.

In addition to the termination and resumption handler responses, exception mechanisms might allow explicit *retrial* of the signaller after taking remedial action, causing re-execution of the entire abstraction with which the handler is associated. The retrial handler response is a rare facility. To the best of this authors knowledge, the only significant programming language that supports it is Mesa [100]. However, retrial is included as a handler response in

the mechanisms described by Yemini (which also supports termination and resumption) and Cocco [107]. A possible reason for the rarity of retrial is the fact that it is that it is implementable within the bounds of the termination model. It is simple for a terminating handler to manually re-invoke the signaller if necessary, or cause the original invoker to do so by side effect. In addition, there is an argument against retrial in general in that there can be difficulty in avoiding an infinite loop of executions and retrials in cases where the exceptional situation cannot be remedied.

Exception handler binding and scope

Exception handler *binding* is the process of associating a particular handler with an exception, instance of an exception, or class of exception. Most mechanisms have *semi-static* handler binding, meaning that for a particular exception, the handler is statically associated with the abstraction from which an instance of that exception may dynamically propagate. Since different handlers may be associated with a particular exception in different contexts, the actual handler invoked is not fixed for each exception and some form of dynamic lookup is necessary. This is the case with the *try* and *catch* binding constructs of Java and C++, where the same exception type may be handled differently (caught) in different guarded blocks (tried). Many mechanisms allow *default* exception handlers, possibly used in conjunction with other handlers, which are capable of handling any exception. This is useful when an abstraction may raise several exceptions, and a single handler would suffice for some or all of them. As mentioned before, a more general mechanism is exhibited by C++ and Java, in which exceptions are normal objects. A subtyping hierarchy allows a handler to catch any exception objects that are subtypes of the statically specified exception type. C++ and Java catch exceptions by stating their willingness to handle a particular exception *type*. Type ambiguity among several viable handlers is resolved by the static ordering of handlers in the program source.

Fully *dynamic* exception binding mechanisms sacrifice static safety for flexibility. For example, the Windows NT operating system [108] provides an exception handling mechanism where handlers controlled by the operating system are invoked dynamically by message passing. Although this mechanism and other operating system based mechanisms can be unreliable due to the lack of static checking, they have the advantage of providing a uniform interface for all programming languages and thus allow the propagation of exceptions across process boundaries. AML/X [109] is the only programming language (as opposed to operating system) that has a dynamic binding mechanism. However, this is perhaps not a deliberate decision on the part of the language designers, but an artefact of the language's dynamic scoping, which allows the run-time determination of identifier's bindings.

With *statically* bound mechanisms, the particular handler that will be bound to an exception instance is determined at compile time. Yemini's mechanism and Knudsen's sequels are both statically bound, and the raising of an exception syntactically and semantically resembles a procedure call. Thus, there is less emphasis on exceptions as distinct entities. Yemini's mechanism is of particular interest, since although it is entirely statically typed, it allows the full range of handler responses (termination, resumption, and retrial).

The granularity of handler association, or handler *scope*, is usually set at the procedure or block level. This is a static association between the handler and the abstraction it guards. For example, in C++ and Java the handler scope has explicit evidence of handler affiliation with an exception activation point – there is an explicit association between try and catch blocks. Depending on the exception handling mechanism, however, handlers can sometimes also be associated with an expression, an object, or a process. Allowing the association of handlers with expressions, although general, tends to reduce program structural coherence due to the embedding of exceptional code (handlers) within the block and procedure 'logical units of computation'. Yemini's mechanism allows expression handler scope. However, her mechanism is designed more as a proof of concept of a general, orthogonal, statically typed exception handling mechanism, and does not emphasise textual separation of exceptional code from normal program code.

Some exception handling mechanisms allow *local handler scope*, where the exception is handled within the abstraction that raises it. For example, Ada allows an 'except' statement to be placed within a procedure that handles exceptions raised within that procedure. Clu has a similar facility. Local handler scope is of questionable value, since it is less of a mechanism and more of a syntactic sugar over flow of control with conditional expressions. However, it does allow the explicit separation of exceptional code and normal program logic, which normal flow control cannot achieve.

Exception handling in C++

In C++, exceptions can be of any type, including primitive values, objects, and pointers or references to objects or values. However, a particularly common methodology is to derive all exceptions types from a common *Exception* base class and raise only references to exception objects.

Any number of handlers can be statically bound to distinguished program blocks (try blocks). Handler selection is achieved by dynamic lookup of the exception type in the handler list (catch blocks). Thus, C++ has semi-static handler binding. Since C++ allows subclasses, the possibility of ambiguity in handler selection arises. This is resolved by appeal to the static ordering of handlers in program text.

Exceptions can be explicitly raised with the throw keyword, and are automatically propagated up the dynamic invocation chain terminating each activation frame along the way until they are handled. If no handler can be found, the program itself is terminated. C++ allows functions and class methods to be augmented with an exception interface, which specifies the types that can legally be propagated by them. By default, the program is terminated before an exception can be propagated illegally, but this behaviour can be reprogrammed if necessary.

Consider the following example.

```
class MyException : public Exception {
public:  int data;
};

void f() {
        try {
                //do do something that might raise an exception
        }
        catch(MyException& e) { /*handler for MyExceptions*/ }
        catch(Exception& e) { /*handler for Exceptions*/ }
        catch(...) { /*default handler that will catch all exceptions*/ }
 }
```

If an exception is raised from the try block, its type will be checked against each of the handler signatures in order. If a match is made, a binding is made between the exception and the identifier in the handler signature, and the handler code executed.

The Java exception handling mechanism is very similar to that of C++. The main difference is that Java statically enforces handling of exceptions named in a method's exception interface. However, specification of exception interfaces is optional.

Exception handling in Clu

The arguments that handler resumption and unrestricted exception propagation are undesirable properties for an exception handling mechanism are compelling [55][106]. The Clu programming language [110] incorporates an exception handling mechanism with handler termination only, and single level propagation. The other properties of Clu's exception mechanism can be summarised as follows.

- Semi-static handler association with single procedure invocations or blocks of arbitrary size, and default handlers are allowed.
- Exceptions are typed identifiers and are parameterisable.
- Procedures must specify in its interface the exceptions that it can signal.
- Handlers may be local scope.

Clu does not automatically propagate exceptions through multiple levels of abstraction and all exceptions must be named in procedure interfaces. This means that it is possible to statically check that all exceptions are handled at the level above that from which they can be raised. However, the language does not enforce this. Instead, it dynamically converts unhandled exceptions to the special *failure* exception. This policy was adopted based on the argument that it can be proved that some exceptions will not be raised from a procedure, even if they are specified in the procedure interface. For example, the possibility of raising a divide by zero exception can be eliminated by explicit checks in program logic, so there need be no associated handler. The possibility of raising the failure exception need not be specified in procedure interfaces, and the failure exception is automatically propagated. However, default handlers will catch it. Consider the following example.

```
power : proctype( real, real ) returns ( real )
        signals( zero_div, complex_result, overflow, underflow )
...
begin
  ...body of block containing invocations of power...
end
except  when zero_div : ...handle zero division...
  when overflow, underflow : ...handle either exception...
  others : ...all other exceptions (complex_result and failure)...
end
```

Program blocks may contain two or more procedure invocations that can raise the same exception, but that exception must be handled differently in each case. The fact that only a single handler for that exception can be associated with the block can lead to problems in specifying appropriate flow control. For example, in the program fragment above, if there are

two invocations of power, the significance of zero_div may be different for each. However, only one handler can be written. In part, this motivates the inclusion of a mechanism for local handler scope, where exceptions must be handled in the routine itself and cannot be propagated. A procedure that encounters an exceptional condition can handle it directly, within the exception mechanism but without informing the invoker. This allows the procedure to mask the exception occurrence if it is thought that the invoker would not be able to handle the exception, or it is not appropriate for it to do so. If the local finds that it cannot resolve the problem, then a different exception can be signalled and propagated to the invoker.

The exception handling mechanism in Clu is more constrained than most, and because of its simplicity is generally considered to be more conducive to well-structured programs. However, it can be argued that in the case of local scope handlers, Clu fails to provide sufficient textual separation between the standard execution specification and the exceptional execution specification.

Sequels

The *sequel* is a language construct due to Knudsen [111][112], and derives from a similar construct developed by Tennent [113]. Knudsen proposes a mechanism that unifies exceptions and handlers, and can statically determine which computations should be terminated on the raising of an exception.

Sequels are similar to procedures, except that after execution a sequel transfers control to the termination point of the block in which it was *declared* instead of the command following its invocation. That is, successful execution of the sequel results in immediate termination for the block enclosing the declaration of the sequel. This is in contrast with most exception handling mechanisms, where the termination level of an exception is a property of the raising statement. Since a sequel may be declared in any outer scope, the mechanism allows multi-level 'propagation'. However, the target handler is always statically bound to the exception invocation. Sequels are statically typed exceptions, unified with their handler, and are raised in a manner similar to procedure invocation.

Sequels may be *prefixed*. This allows the exception handling flow of control to pass directly to the sequel of the termination level sequel, which can perform some pre-processing of the exception. Control flow then passes to the sequel of the next inward block, which may itself perform pre-processing before the next inward block, and so on until the innermost prefixing sequel (the one originally invoked) is executed. The recursive chain is then unwound back to outermost sequel, each sequel being given the opportunity to perform post processing, until finally control passes to the outermost prefixing sequel, which performs its cleanup and terminates at the level of its enclosing abstraction. Knudsen states that,

> "For...exception handling within a specific block to be secure and well-behaved, one must assume that the outer blocks are in a consistent state. If such consistency is not ensured, then it is difficult to handle exceptions because the block cannot assume anything about the state of outer blocks." [111]

Knudsen presents the following example of how the termination properties of traditional exception handling mechanisms can be problematical. On the raising of an exception, if the termination level is more than one level of scope outward[1], then it is possible that one or more blocks must be in a particular state before consistent termination of the inner blocks can be ensured. The primary example of this is the opening of a file to which the inner blocks write as part of exception processing. It is important here for the cleanup actions of each intermediate block to depend upon the initially generated exception.

Prefixed sequels are similar in purpose to Mesa's *unwind* operation, where, if a handler decides to terminate rather that resume the propagating abstractions, executing an unwind allows each activation in the propagation chain to perform a cleanup operation before termination. In Mesa, however, the propagation chain is dynamic, and not static as with sequels. This means that in Mesa there is no static enforcement of policy with respect to either the use of unwind, or presence of cleanup code at each abstraction level. Therefore, when calling another abstraction, an activation might expect to be terminated without notice by an unhandled exception. Good programming methodology can prevent this kind of behaviour, but methodology is inherently weaker than static enforcement.

The concept of *virtual* sequels is introduced in order to account for irregular behaviour when directly calling sequels that are not associated with the current scope level. That is, invoking prefixed sequels in outer blocks leads to the abrupt termination of all inward blocks with respect to the declaration of the invoked sequel.

Even though the sequel mechanism is entirely statically typed, it allows for flexible exception handling. However, it can be argued that in practice, the mechanism is somewhat complex, and the recursive chaining of multiple overlaid prefixed virtual sequels can be difficult to grasp. Moreover, the motivational example presented (file write during exception handling) may be considered somewhat artificial, and possibly does not demonstrate a general

---

[1] This also applies to mechanisms that do not allow multi-level propagation of a single exception, since each level may have a handler that raises a new exception.

requirement for an entire mechanism. Finally, the tight coupling between abstraction levels that results from the use of multi-level propagating sequels compromises modularity, and is in direct conflict with the design methodology adopted in Clu.

Summary

There is a need to propagate information from the operation that detects an error up to a higher level of abstraction so that the error can be handled in an appropriate context. This can be achieved by overloading the function return mechanism to carry error information, possibly in conjunction with global state. However, this methodology has several drawbacks:

- Reduced structural coherence due to explicit tests and associated flow control.
- It cannot be captured in function interfaces and must be documented separately.
- Use of global state compromises concurrency and structural coherency.
- Conformance is not enforced, increasing likelihood of programming errors.
- It is not typed, making the writing of correct programs more difficult.
- It is difficult to reason about formally.

Exception handling mechanisms provide a language-level alternative to this methodology, automating the process of error information propagation and handler selection. This means that programmers are left only with the responsibility of detecting the original error, and writing an appropriate recovery routine. With exception handling, error recovery policy can be modularly separated from computational logic. In conjunction with the fact that the programmer need not write explicit error propagation code, this greatly improves the clarity of program source. There are four main orthogonal aspects of exception handling mechanisms, each of which can take more than one form.

- *Flow control* – the flow of control applied to the operation raising an exception that is taken on handler activation can be one or more of termination, resumption, and retrial. Retrial is uncommon, and is to an extent subsumed by termination. Resumption is intended to prevent loss of computation up to the point of detecting an error, but has several drawbacks, the most important being loss of modularity. Termination retains modularity, and is generally seen as the most appropriate behaviour on handler activation.

- *Exception form* – exceptions can be types, identifiers, values, or objects. In general, the more information an exception can carry with it, the more flexible the exception handling mechanism. Ada has an inflexible mechanism, since its exceptions are simple identifiers that carry no extra information. In contrast, Java exceptions are arbitrary objects. Object-oriented languages with exceptions as objects have the additional advantage of allowing the handling of more than one exception with a single handler, by inclusion polymorphism.

- *Propagation* – when an exception is raised, the number of abstraction layers through which it can be automatically propagated before handling is significant. In general, this is either a single level, or any number of levels. Single level propagation incurs syntactic and logical overhead, but is seen as a more modular alternative to multilevel propagation. It also makes possible a greater degree of static checking. Despite the consensus on the benefits of single level propagation, the contemporary programming languages Ada, Java, and C++ all allow multilevel propagation.

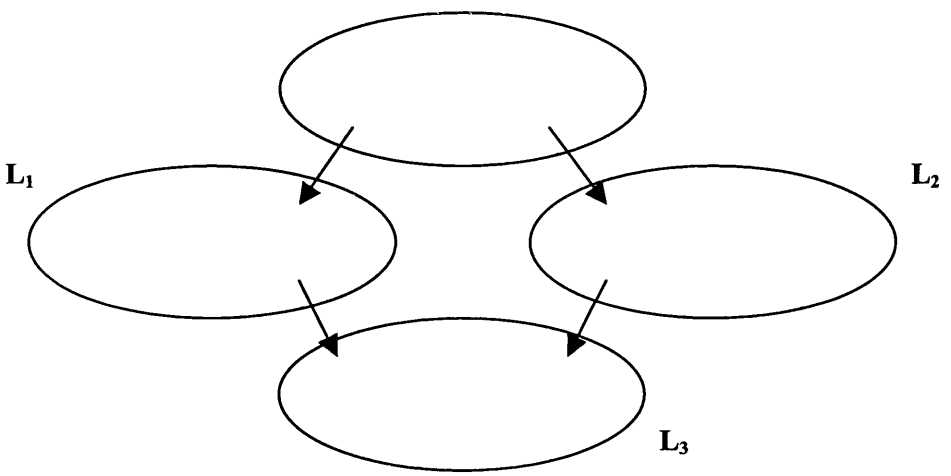
- *Interface and binding* – static exception handling mechanisms bind exception occurrences to their handlers at compile time. The interface resembles a procedure call and there is less emphasis on exceptions as entities. Semi-static mechanisms are the most common. These statically associate handlers with abstractions from which exceptions may propagate, commonly program blocks. In some cases the exception interface might require all possible exception propagations to be specified explicitly. Although this can ensure that all exceptions are handled, dynamic checking is still required in order to select the appropriate handler for a particular exception, since different handlers may be associated with the same exception in different contexts. Purely dynamic mechanisms are primarily operating system based, and although they are flexible and allow exceptions to cross inter-process boundaries, they are untyped and so unreliable.

9: Formal Issues

In this chapter, we present some formal concepts. First, we show that supervisors conceptually contain the service combinator algebra, by implementation. Second, we describe an implementation of the supervisor environment model that is more efficient than the obvious naïve approach and prove its correctness.

Conceptual containment of Service Combinators by supervisors

Given a basic language L, that includes Web fetch as a primitive operation, there exists a class of language L_1 that adds the service combinator (SC) abstractions to L. There is also a class of language L_2 that adds persistent relative observables (PRO) and supervisors (S) to L, and a class of language L_3 that is a combination of L_1 and L_2. In this framework, the language L_2 is Focus.



Now we will show the implementation of abstractions written with L_2 that directly provides the functionality of the service combinator algebra extensions of L_3. Our implemented 'algebra' provides individual supervisor abstractions in a one to one mapping with the original algebra and has minimal syntactic overhead. Since L_2 can simulate L_3 directly, we show that L_2 is conceptually equivalent to L_3 and so subsumes L_1.

To understand the implementation, it is important to remember that the supervisor construct and the service combinators both have parameter passing semantics that are not eager. With these semantics, expressions are passed into enclosing abstractions before evaluation, or in the case of supervisors, during evaluation. For example:

```
limit( 2, 3000, ( url( "http://a.org/" ) | url( "http://b.org/" ) ) )
```

Here, the limit is logically applied before the fetch of the URLs, which is in contrast to parameter passing mechanisms that evaluate eagerly.Now we present the implementation of service combinators with supervisors. The basic service results in a string:

```
type Service is string
```

Although the Focus type system allows a variety of mime types, we concern ourselves only with html documents here in order to keep the program fragments concise. The sequential combinator evaluates the primary service first, and if it fails returns the secondary service instead:

```
let seq = supervisor(primary:Service; secondary:Service → Service) is {
  suspend secondary
  waitfor done primary or failed primary
  if done primary then primary else {
        activate secondary
        waitfor done secondary or failed secondary
        secondary
  }
  primary
}
```

Note that although the implementation immediately suspends the secondary service, there can be no computational interference from update by the secondary before this occurs. This is because each thread is isolated within its own environment. In any case, the service combinator algebra is declarative since services contain no side effect. The suspension and, if necessary, activation of the secondary service serves only to provide a pattern of computation close to sequential execution in terms of its temporal behaviour.

The concurrency combinator returns the result of whichever service completes first, when both are executed concurrently.

```
let con = supervisor(a:Service; b:Service → Service) is {
  waitfor not active a or not active b
  if done a then a
  elseif done b or failed a then b
  else a
}
```

The con supervisor waits for either a or b to become inactive. Whichever thread becomes inactive first has either completed or failed, so we check for completion (done) of both threads. If neither has complete, the original inactive thread must have failed, so we force reliance on the other. Failure of that thread will result in failure of the supervisor.

The implementation of the timeout combinator is shown below. It uses a thread that sleeps for a known absolute time in order to determine when to infer failure of the service:

```
let timeout = supervisor(millisecs:int; s:Service → Service) is {
  let timer = supervisor (s:Service; sleeper:void) is {
        while not done s and not failed s do
              if done sleeper do fail
        s
  }
  timer(s, sleep millisec)
}
```

In the inner timer supervisor, if the thread s fails then the return of the value of s will cause the propagation of failure out of the timeout supervisor. The repeat combinator repeatedly invokes the service until success:

```
let repeat = supervisor(s:Service → Service) is {
  while not done s do {
        while active s do { }
        if failed s do retry s
  }
  s
}
```

The stall combinator does nothing forever.

```
let stall = function(void → Service) is { waitfor false; "foo" }
```

The fail combinator is defined trivially, since it maps directly onto a primitive construct of the supervisor mechanism.

```
let scFail = function(void → Service) is { fail; "foo" }
```

Following are global variables for the rate limit and startup time of URL fetches. Since there may be several limits in an expression, different values for rateLimit and startup may apply for different URLs. We will use thread update exposure, controlled by the expose operation, to ensure the appropriate limit context for each URL in a combined service.

```
let startup = loc(maxfloat())
let rateLimit = loc(maxfloat())
```

We must pass a value for rate and time into the limit combinator. However, if we were to pass simple floating point values[1] in, the supervisor body for limit would not be able to pass their evaluations to the service. This is because the computational context of supervisor bodies is isolated from those of the threads it supervisors. However, the rate constraint and startup time need to be propagated to all URL fetch services in the nested service so that, in a sense, the URL fetches can limit themselves. To achieve this, we must take a different

[1] More accurately, we pass in simple threads that compute floating point values.

approach. Supervisor bodies (in this case limit) cannot expose themselves to parameter computations. This means that the evaluation of any particular parameter thread cannot be communicated to another parameter thread by the supervisor body. However, we note that the supervisor body for the limit combinator does not need to know what the rate constraint and startup time actually are. Instead, it only needs to communicate those values to the service computation. Our solution, then, is that the act of passing the rate and startup parameters to the limit combinator causes a side effect to the global variables above. The supervisor body can then expose this side effect to the service. We achieve this by declaring an abstract type that causes side effect to the locations above whenever an instance of that type is created. The fact that we use abstract data types ensures that only these side effecting values can be passed to the limit combinator. The abstract data types are defined as follows:

```
type Time is abstype {
  private
        type Time is float
  public
        createTime : function(t:float → Time) is {
              if t < at startup do startup := t; t
        }
}
```

In order to model the service combinators semantics for nested limits (unification according to most constrained values), the creation of the Time and Rate values will overwrite the existing global variables only if they are less than the existing values. Otherwise they remain unmodified.

```
type Rate is abstype {
  private
        type Rate is float
  public
        createRate : function(r:float → Rate) is {
              if r < at rateLimit do rateLimit := r; r
        }
}
```

An invocation of limit might look something like:

```
limit( createTime(5), createRate(1000), url( "http://foo.org" ) )
```

Now we show the implementation of the limit combinator. Limit waits for the specified startup and rate threads to evaluate, both of which may have side effect as described above. It then exposes these updates to the service. Although the limit variables are global, the semantics of expose with respect to environment copying means that any *new* values for rateLimit and startup are visible *only* to the parameter service.

```
let limit = supervisor(startupSec:Time; bytesPerSec:Rate; s:Service
        → Service) is {
  waitfor done startupSec and done bytesPerSec
  expose startupSec
  expose bytesPerSec
  expose s                    //let s 'see' the limits
  s
}
```

In the service combinator algebra, url is the most primitive service. Individual invocations of url are themselves responsible for inferring failure should a constraint imposed by the limit combinator be violated. The combinator uses a supplementary supervisor to get a handle on absolute time for the startup period.

```
let url = function(url:string → Service) is {
  let downloader = supervisor(fetchThread:Service; startupThread:void) is {
        while active startupThread do
              if failed fetchThread do fail  //Abort on absolute failure
        while not done fetchThread and not failed fetchThread do
              if rate fetchThread < at rateLimit do fail
        fetchThread
  }
  downloader(get url, sleep at startup)
}
```

The service combinator expression:

```
limit( 5, 2000, ( url( "http://a.org/" ) | url( "http://b.org/" ) ) )
```

would be written in our implemented algebra as:

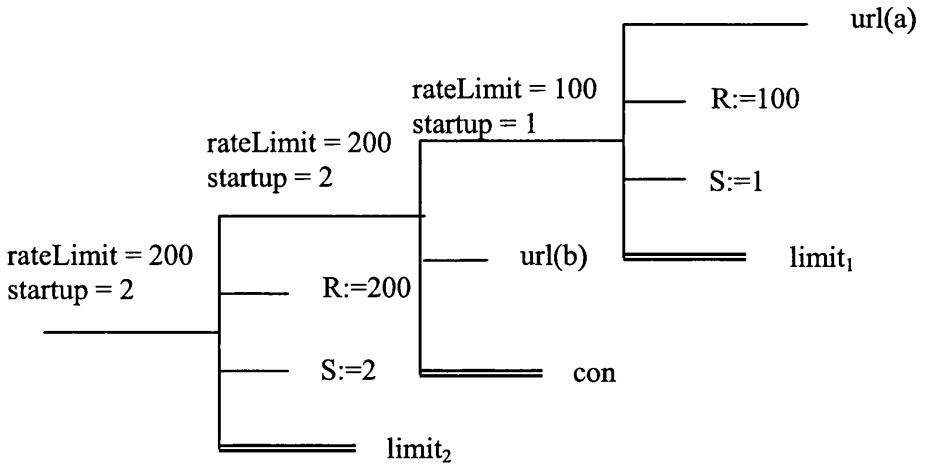
```
limit( createTime( 5 ), createRate( 2000 ), con( url( "http://a.org/" ),
        url( "http://b.org/" )))
```

The following expression, shown in service combinator syntax for brevity:

$$limit_2(1, 1000, (url("http://b.org/") | limit_1(2, 2000, url("http://a.org/"))))$$

has the thread tree shown below, after the exposures by the limits. The tree has environment annotations to show contents of the rateLimit and startup locations. Particular values for rateLimit and startup apply only to their subtrees.

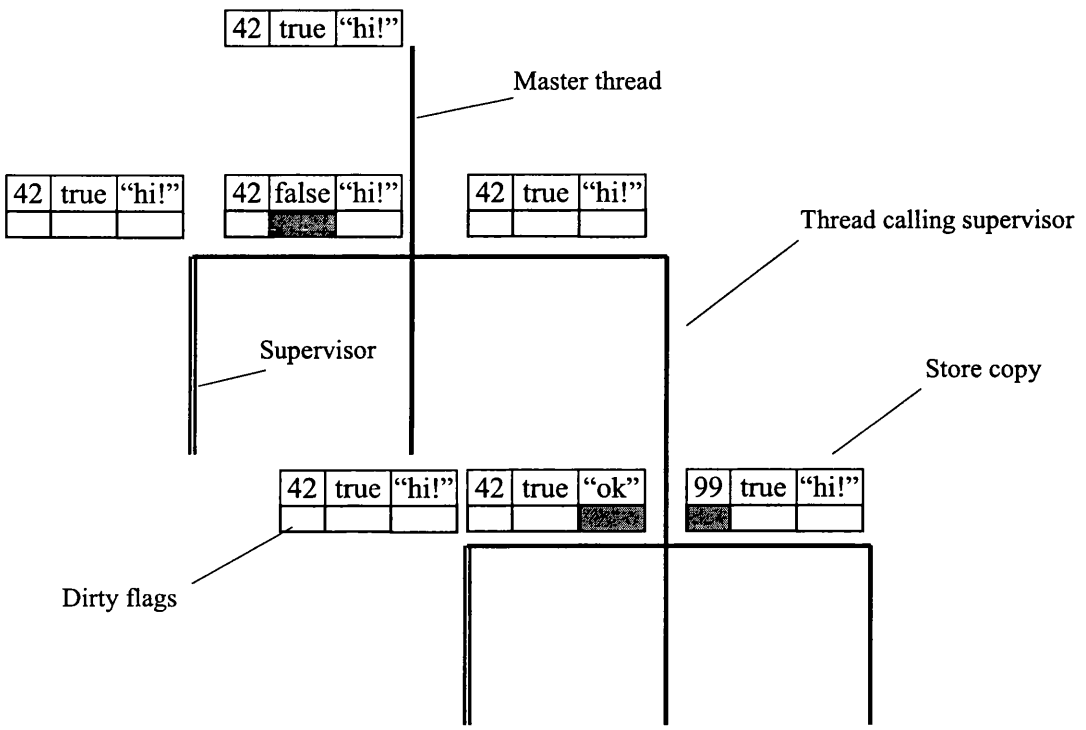
Note that service combinators have the 'observables' of rate, time, and latency, which are the same as those of Focus. However, Focus has the additional historical context for each and the additional probability observable.



Supervisors are probably not conceptually equivalent to the service combinators provided by WebL, since it is difficult to directly model the propagation of exceptions and the unconstrained update allowed to WebL services. However, we argue that the presence of unconstrained update is what makes WebL service combinators a weak abstraction when compared to the more constrained supervisor environment model. In addition, we argue that WebL's integration of an exception handling mechanism with service combinators weakens the latter. For these reasons, we did not see the merit in attempting a proof of conceptual equivalence by implementation, since it is unlikely that one would wish to implement the combinator semantics of WebL.

Implementing the Supervisor Environment Model

The Focus abstract machine's execution model is based upon a tree, where every leaf node represents an executing thread and every internal node is a suspended 'parent' thread that has invoked a supervisor. Thus, branching of the thread tree indicates concurrency. An execution cycle of the abstract machine is a recursive traversal of the tree, executing a single instruction for each active thread – the leaf nodes. The root of the tree we term the master thread; its failure implies failure of the entire execution. Failure of sub-threads, however, may or may not result in failure of the program, depending on how their failure is propagated up the tree. This failure propagation is dependent upon the programmed failure semantics of the



supervisor threads. Thus, only failure of the supervisor results in failure being propagated up the tree. Failure of a supervisor is likely to be a result of a supervisors inability to recover from observed or interpreted failure of supervised threads.

To naively implement the semantics of the supervisor environment model the entire active state space of the abstract machine must be duplicated for every thread inception, including supervisor body threads. The usage of the term *environment* here distinguishes thread's state space copies from a normal store in that it refers to a store where each location has a 'dirty' flag, indicating the occurrence of update. In the naïve solution, this large amount of copying is required because locations are first class, and so no static reasoning can be achieved with respect to which locations are accessible dynamically by a thread. Any thread exposure mandated by a supervisor results in the unification of environments for the exposed thread

and the parent thread, which originally invoked the supervisor. This is followed by the recapture and duplication of the unified environment into the exposed thread. An example outline of the dynamic tree structure is presented in the diagram above.

This naïve solution implements the high-level supervisor environment model as required. It does, however, incur intolerable execution overhead for several reasons:

- The state space may be extremely large. This is in tension with the fine-grained granularity of concurrency in the model as it implies excessive copying.
- A majority of the state space will, in general, not be accessed by a thread, and as such need not be duplicated within thread local environments.
- Environment unification requires a search across dirty flags, linear in the size of the state space.
- Thread exposure can be arbitrarily frequent, incurring the environment unification and copying overhead each time.

A more efficient solution lies in the use of read and write *shadow stores*, which are analogous to caches. Each thread has two shadow stores, which contain logical location identifiers and the values within them *with respect to the execution of that thread.* That is, a single location can be present in many different shadow stores at one time, with a different associated value in each case. However, locations can only be 'truly' updated by the master (or top level) thread. This is consistent with sub-tree threads *shadowing* the main store.

The write store consists of location-value pairs that have been written by the thread. If a thread never updates a location, that location will never be present in its shadow store. The read store also consists of location-value pairs, but is never written to by its associated thread. Only the immediate supervisor can cause writes to this store.

During execution, instructions may make three types of request of threads with respect to locations. Firstly, location *inception* results in a location being created and inserted into the write cache of that thread, uninitialised. The location is statically guaranteed to be initialised at a later point[1]. Secondly, location *dereference* causes the thread to first query its write cache for the requested location. If present, the location value from that store is returned. Otherwise, the read cache is checked in the same way. If present in neither store, the request is delegated

[1] In principle, the thread may be terminated before initialisation, but in this case the location will reside only on the stack of the dying thread.

to the parent thread. This can continue recursively until reaching the master thread. Finally, location *update* causes the thread to place the location and assigned value into its write cache, overwriting any value already present.

Another significant event that occurs in the system is thread *exposure*, directed only by supervisor threads. On exposure, a thread performs a logical location *update* to its parent thread for each location in its write store. Further action is required in addition to the write store insertions in that each insertion causes a *PushDown* operation to be applied to all sibling threads at that level in the tree. PushDown is parameterised by a location and value pair. The value in this pair is that which is *overwritten in the insertion to the parents read store*, or the 'original' value of the location with respect to the parent thread, before exposure. If the location is not in the parent's write cache to be overwritten as part of the logical update, then it must be read according to a normal dereference operation – the value may be present in the read store or further up the tree. All threads receiving a PushDown operation insert the location and value parameter pair into its read store. Importantly, this read store insertion does not overwrite location value pairs in the read store if they are already there. After all PushDowns have been resolved, the exposed thread purges both its read and write shadow stores, and continues execution as normal.

The final event that causes action by the run-time environment model is that of thread *retry*. Retry causes the thread to purge its write store, but not its read store, terminate any subthreads it may have, and reset its program counter and environment to the point of inception.

We now demonstrate the correctness of this algorithm by proving that the optimised environment model is equivalent to the naïve environment model. Our proof is constructed with the functional programming language Haskell [114], and is based on well-known techniques, a good introduction to which is provided by Thompson [115]. There are several different formalisms that we can use, but we choose Haskell because our algorithms can be syntax and type checked, and even executed to provide reassurance that our approach is correct at each stage. Haskell is often used for proofs of algorithm correctness, and has many other benefits in this context [116]. We are interested in the fact that since it is a lazy functional language, infinite lists can be expressed. We use infinite lists to model environments and sequences of abstract machine instructions. Doing so is simpler than being forced to declare dynamic structures that model the store as it grows, had we based our proof directly on the implementation. In addition, Haskell is a pure functional language, and so has referential transparency. The absence of update allows us to reason with programs easily.

Briefly, in Haskell capitalised identifiers refer to types, the symbol '::' means is of type, square brackets denote lists, '++' is list concatenation, ':' is an infix list construction operator,

and parenthesis denote tuples. We will describe other features (in particular standard functions) of the language as we encounter them. The 'data' keyword is a type specifier analogous to a union or variant type. A value of the type may arise from one of two or more different constructors. The functions fst and snd project the first and second values in a tuple.

Supporting Definitions

These are the types we are using to represent the various elements of thread trees:

```
type Value = Integer                    --values are simple integers
type Location = String                  --location identifiers
data Op =       Create Location Value |             --create a new location with given value
                Move Location Location |            --updates one location with value from another
                InvokeSupervisor CodeSeq [CodeSeq] |        --invoke a supervisor
                Expose Int |                        --expose thread index
                Nop                                 --a null operation
type CodeSeq =  [Op]
type Continuation = CodeSeq             --code executed after supervisor completes
type Store = [(Location, Value)]        --Stores are lists of location value pairs
type Env = (Store, Store)               --environments are store tuples
```

The next type, Thread, represents the actual thread tree:

```
data Thread =   Thread Env CodeSeq |
                Supervisor Env Thread [Thread] Continuation
                --this is: Supervisor environment bodythread supervisedthreads continuation
```

That is, a thread tree is either a single Thread with its associated environment and code sequence, or a supervisor with environment, distinguished thread (body), a list of supervised threads, and a continuation code sequence. The supervisor variant type represents the fork in a thread tree. Supervisor as a concept here is distinct from the supervisor body, and is a more abstract entity into which a thread 'converts' on supervisor invocation. Thus, the Supervisor environment is really the parent environment, the environment of the invoking thread, and the continuation is the remaining code sequence for that thread. The supervisor body is itself a thread, with its own environment and code sequence.

Environments are used differently by the two algorithms. The naive algorithm uses the fst Store as a copy of the entire system store, and locations in the snd Store are dirtied locations. Location lookups first check snd, then fst. This is equivalent to a dirty flag mechanism. The optimised algorithm uses fst as its read store, and snd as the write store. Due to the similarity here, several functions can be shared between the naïve and optimised algorithms, easing our proof of equivalence. Now we a define a set of supporting functions. First we have Thread

field extraction functions. We use these so that we do not have to distinguish between Threads and Supervisors in the algorithms, which simplifies matters:

```
-- Thread field extraction functions
getOps :: Thread → CodeSeq
getOps (Thread _ ops) = ops
getOps (Supervisor _ _ _ ops) = ops

getEnv :: Thread → Env
getEnv (Thread e _) = e
getEnv (Supervisor e _ _ _) = e

setEnv :: Env → Thread → Thread
setEnv env (Thread _ ops) = Thread env ops
setEnv env (Supervisor _ body threads cont) =
        Supervisor env body threads cont
```

Now we define a set of store functions. These perform operations on stores, such as update and location insertion, that are used in the main algorithm. Even though environments are used in different ways by the two different algorithms, many of these store functions are used in both.

isLocInStore returns true if loc is present in the Store:

```
isLocInStore :: Location → Store → Bool
isLocInStore loc [ ] = False
isLocInStore loc store
        | loc == fst (head store)       = True
        | otherwise                     = isLocInStore loc (tail store)
```

storeUpdateLoc modifies a location in a store, or inserts it if it is not present:

```
storeUpdateLoc :: Location → Value → Store → Store
storeUpdateLoc loc val [ ] = [(loc,val)]
storeUpdateLoc loc val ((l,v):rest)
        | loc == l          = (loc,val):rest
        | otherwise         = (l,v):(storeUpdateLoc loc val rest)
```

storeUpdateStore performs a series of storeUpdateLocs to a target for all locations in a source store:

```
storeUpdateStore :: Store → Store → Store
storeUpdateStore [ ] target =  target
storeUpdateStore ((loc,val):rest) target =
```

Aside

storeUpdateStore is essentially a biased union. The target store (second parameter) is unified with the source store, with duplicates being resolved by location bindings in the source store overriding those in the target. This function is used frequently in the proof that follows, so we define an infix operator '»' to represent it more concisely. It may be helpful to think of » as representing a flow of location bindings from left to right.

```
source (») target = storeUpdateStore source target
```

```
        storeUpdateStore rest (storeUpdateLoc loc val target)
```

storeInsertStore updates a target store with all locations in a source store, but only if those locations are *not already present in the target.* An observation that is critical to the final stage of the later proof is that this can be defined in terms of storeUpdateStore (»):

```
storeInsertStore :: Store → Store → Store
storeInsertStore st target =
        target » st                    --the stores operands are reversed from storeUpdateStore semantics
```

storeDeref returns the value of a location in the store. Our model requires that the location is present: This function uses a fold, which is an abstraction over recursion. The details of fold are not especially important here, other than it can represent recursion as a single function. The interested reader is referred to the literature [114].

```
storeDeref :: Location → Store → Value
storeDeref loc store =
        foldr (\(loc',v') v → if loc' == loc then v' else v ) undefined store
```

storeDerefStore takes all the locations from a store, and looks up their value in another store, returning the store with the other values for the location

```
storeDerefStore :: Store → Store → Store
storeDerefStore src target = [(loc, storeDeref loc target) | (loc,_) ← src]
```

Now we define some functions over environments, most of which make use of the store functions. Both naïve and optimised algorithms use many of these environment functions, since although the environments are used in different ways they have the same structure and so have similar primitive operations over them.

envUpdate modifies a location in the dirty (or write) Store, or inserts it if it is not present:

```
envUpdate :: Location → Value → Env → Env
```

```
envUpdate loc val (s1,s2) = (s1, storeUpdateLoc loc val s2)
```

expose performs a series of updates from one environment to another for dirty (write store) locations only:

```
expose :: Env → Env → Env
expose (_,s) targetEnv = foldr (\(loc,val) acc → envUpdate loc val acc) targetEnv s
```

capture returns an environment with dirty (write store) locations overwriting those in the fst Store:

```
capture :: Env → Env
capture (locs,dlocs) = (dlocs » locs,[ ])
```

envDeref looks up a location in the given environment, checking the dirty (write) Store first. It merges the dirty and non-dirty stores, dirty overriding

```
envDeref :: Location → Env → Value
envDeref loc (locs,dirty) =
        storeDeref loc (dirty » locs)
```

Algorithm Definitions

Now we can write down a *step* function for the naïve algorithm. step executes a single operation for every thread in the thread tree.

```
step :: Thread → Thread
step (Thread env [ ]) = Thread env [ ]                    -- stepping a completed thread does nothing
```

Our first non-trivial step follows. This executes an active basic thread (not a supervisor). The function takes the first operation from the instruction stream, inspects its value and acts accordingly.

```
step (Thread env (op:ops))      = case op of
    Create loc val →
        Thread (envUpdate loc val env) ops                -- add the loc and val to environment
    Move readLoc writeLoc → let
        val = envDeref readLoc env in                     -- lookup the value
            Thread (envUpdate writeLoc val env) ops       -- update location and throw away move op
    InvokeSupervisor bodyCodeSeq threadCodeSeqs → let
        body = Thread (capture env) bodyCodeSeq
        threads = [Thread (capture env) codeSeq | codeSeq ← threadCodeSeqs] in
            Supervisor env body threads ops
    Nop → Thread env ops
    Expose t → undefined                  --not permitted for a normal thread, Focus has static restrictions
```

This next step completes execution of a supervisor, turning it back into a normal thread. Remember that the supervisor environment (env) is that associated with the thread that invoked the supervisor and not the supervisor body.

```
step (Supervisor env (Thread bodyEnv []) _ cont) =                    -- pattern match on empty body CodeSeq
    Thread (expose bodyEnv env) cont
```

This next step executes a single instruction in the body of a supervisor and each of its threads. The result is the same for all cases except where the supervisor body executes an expose operation. We cater for this explicitly.

```
step (Supervisor env body threads cont) =
    case body of
        Thread bodyEnv ((Expose t):ops) → let
                (front,(thread:rear)) = splitAt (t-1) threads      --get at the threads
                e = getEnv thread                                  --the env to be exposed
                newEnv = expose e env                              --expose it to parent env
                newBody = Thread bodyEnv ops                       --throw away expose op
                exposedThread = setEnv (capture newEnv) thread     --new thread after exp
                newThreads = (front ++ [exposedThread] ++ rear) in --remake thread list
                    Supervisor newEnv newBody (map step newThreads) cont
        _ → Supervisor env (step body) [step thread | thread ← threads] cont
```

When stepping a supervisor, we step the body and each of the threads. The result is the same as for a normal thread in all cases (the result of the _ choice in the case statement), *except* when the body executes an expose. In this case, we must resolve a new environment for this point in the tree that incorporates those updates dirtied by the exposed thread. Note that even if the body of this supervisor is itself a supervisor, then the result is the same since that supervisor body cannot expose the threads at this point.

This completes the implementation of the naïve algorithm, which directly implements the intended semantics for the supervisor environment model. Now we present the optimised step algorithm, defined by the function step'. In general, we use an apostrophe to represent functions or values associated with the optimised algorithm.

The optimised algorithm requires a 'push down' operation. This operation is used to insert locations into the read stores of threads that might be affected by the exposure of another thread. To perform a push down, for every location in the exposed threads write store, we insert the original value of it (obtained from the parent environment) into the affected thread's read store. We show the implementation of pushStoreThread here rather than in earlier supporting functions section, because it is only used by the optimised algorithm.

pushStoreThread pushes all locations in the Store into the read store of the given Thread:

```
pushStoreThread :: Store → Thread → Thread
pushStoreThread locs thread = let
    (read, write) = getEnv thread in
        setEnv ((storeInsertStore locs read), write) thread
```

We accumulate a 'store chain' while recursively descending the thread tree. The store chain is an accumulation of locations in the write and read stores of parent environments. In this way, threads that do not have a location in their read or write stores can obtain them from the store chain. In the Focus run time system there is no store chain. Here, we use the store chain to model the fact that in the Focus run time system, threads that read the value of a location not in the their immediate environment 'delegate' the read to the parent thread that activated it. The parent may in turn delegate to its parent, and so on. The store chain represents the search path a dereference would take back up the thread tree. It is an artefact of the model, and represents dynamic behaviour rather than a data structure that is maintained by the Focus run-time system. Actually implementing such a data structure is just as inefficient as the naive algorithm that copies entire environments.

```
-- step' performs a single execution step for a thread tree
step' :: Thread → Store → Thread          -- step' takes a store chain parameter


-- this step' represents "execution" of a completed Thread
step' (Thread env [ ]) _ = Thread env [ ]
```

This step executes a single operation in a Thread. The most significant aspect of it is for the case where the thread executes an InvokeSupervisor operation. This is the basis of our optimisation, since the newly created child threads and the distinguished thread receive *empty* environments. No copying takes place.

```
step' (Thread env (op:ops)) storeChain = let
    (readStore, writeStore) = env
    newStoreChain = storeUpdateStore readStore storeChain in
        case op of
            Create loc val →
                Thread (envUpdate loc val env) ops
            Move readLoc writeLoc → let
                val = envDeref readLoc (newStoreChain, writeStore) in
                    Thread (envUpdate writeLoc val env) ops
            InvokeSupervisor bodyCodeSeq threadCodeSeqs → let
                body = Thread ([ ],[ ]) bodyCodeSeq      --create with empty store (no copy!)
                threads = [Thread ([ ],[ ]) threadCodeSeq | threadCodeSeq ← threadCodeSeqs] in
```

```
                    Supervisor env body threads ops
  Nop →
        Thread env ops
  Expose t → undefined

-- this step' completes the execution of a Supervisor whose body is finished, reverting the Supervisor to a Thread
step' (Supervisor env (Thread bodyEnv []) _ cont) _ =
    Thread (expose bodyEnv env) cont          -- the supervisor body is exposed automatically on completion
```

Now we present the final and most important version of step'. This version executes a single step for a Supervisor body and Threads, and like the corresponding function in the naïve algorithm, it explicitly takes care of the case when the supervisor body executes an expose operation:

```
step' (Supervisor env body threads cont) storeChain = let
    (readStore,writeStore) = env
    newStoreChain = storeUpdateStore (storeUpdateStore (readStore) storeChain) writeStore in
    case body of
      Thread bodyEnv (Expose t:ops) -> let
        (front,(thread:rear)) = splitAt (t-1) threads      --get at the threads
        e = getEnv thread                                  --the environment to be exposed
        newEnv = expose e env                              --expose it to the parent environment
        originalValues = storeDerefStore (getWriteStore e) storeChain          --get previous values
        newBody = pushStoreThread originalValues (Thread bodyEnv ops)          --push down to body
        exposedThread = Thread (capture newEnv) (getOps thread)                --recreate exposed thrd
        newThreads = (map (pushStoreThread originalValues) front) ++           --rest of the threads
                     [exposedThread] ++ (map (pushStoreThread originalValues) rear)
        in Supervisor newEnv newBody [step' thread newStoreChain | thread <- newThreads] cont
      _ -> Supervisor env (step' body newStoreChain) [step' thread newStoreChain | thread <- threads] cont
```

Now we define two functions that run programs (CodeSeqs) to completion for each of the algorithms. We call them go and go'.

```
-- The naïve algorithm
run :: Thread → Env
run (Thread env []) = env
run thread = run (step thread)
go :: CodeSeq → Env
go program = run (Thread program [])

-- The optimised algorithm
run' :: Thread → Env
```

```
run' (Thread env []) = env
run' thread = run' (step' thread [])        --empty store chain created for each step of the tree (not carried over)
go' :: CodeSeq → Env
go program = run' (Thread program [])
```

Proof of Algorithm Equivalence

Now we can begin to formulate a proof of equivalence for go and go'. If we can prove that go and go' are equivalent for all programs, then we can state that the optimised algorithm implements the intended semantics for environments. Induction is the obvious approach.

- Prove that for all programs p, go p and go' p result in the same store.

This requires simultaneous induction over programs and the tree structures that can derive from programs. This is because for a given program, execution of an InvokeSupervisor operation results in a subtree where each thread itself has a program. Under such circumstances, induction is complex. If possible, we would like to eliminate one of the inductions. We can achieve this by tackling the proof a step at a time:

- For some definition of equality over trees t and t' corresponding to partial evaluations of programs with go and go' respectively, show that t = t' at all stages of program execution.

Here, we eliminate the need to perform induction over entire executions of programs. This is because our inductive step is the execution of a *single* operation by all threads in the tree, without regard to the rest of the program. If we can show that step t = step' t' for all t and t' that are equivalent, then we have our result.

To begin, then, we require a method of equating trees generated by step and trees generated by step'. Although these will have exactly the same structure after the same number of steps, the contents of each thread's environment will differ. However, the *observable system state* with respect to each thread should be the same, and this is the basis for our proof. We use this notion in defining a function that produces a canonical representation for trees that can be compared directly.

We need to equate the observable store for every thread, at every level of the trees. We define observation functions, which return the observable system store from the point of view of a particular point in the tree. For the naïve algorithm (NA) we define:

```
obs :: Env → Store
obs (allLocs, dirtyLocs) =
        storeUpdateStore dirtyLocs allLocs      -- overrides allLocs with dirtyLocs
```

For the optimised algorithm (OA) we define:

```
obs' :: Store → Env → Store                    -- takes the storeChain at that point and the current environment
obs' storeChain (readStore, writeStore) =
        writeStore » readStore » storeChain
```

obs' returns the observable system store with respect to the thread that has the environment consisting of readStore and writeStore, and has had storeChain passed to it on its inception. Now we define a function that flattens the tree according to a pre-ordered traversal, producing a list of observable states, one for each thread. Since the OA and NA use environments in different ways, we require one for each:

```
-- flatten trees produced by step
flattenStores :: Thread → [Store]
flattenStores (Thread e _) = [obs e]
flattenStores (Supervisor e t ts _) =
        [obs e] ++ flattenStores t ++ concat (map flattenStores ts)


-- flatten trees produced by step' – we need to produce a storeChain on the way down
flattenStores' :: Store → Thread → [Store]
flattenStores' storeChain (Thread e _) = [obs' t storeChain]
flattenStores' storeChain (Supervisor e t ts _) = let
        newStoreChain = obs' e storeChain in
                [storeChain] ++ flattenStores newStoreChain t ++ concat (map (flattenStores newStoreChain) ts)
```

Note that the definitions of obs, obs', flattenStores, and flattenStores' are derived from the definitions of step and step', and give us our definition of tree equality. We want to prove that for all thread trees t, t' that were created by same program p and an equal number of applications of step, step':

```
flattenStores t = flattenStores' [ ] t'
```

We do this by induction over evaluation of the program, p. For the base case, where the program has just started and no operations have been executed, we have a flat tree structure and we prove:

```
flattenStores (Thread ([ ],[ ]) p) = flattenStores' [ ] (Thread ([ ],[ ]) p)    (base case)
```

Then we prove the inductive step, assuming that:

```
flattenStores t = flattenStores' [ ] t'                                        (induction hypothesis)
```

We prove:

```
flattenStores (step t) = flattenStores' [ ] (step' t' [ ])                     (induction step)
```

Then by induction, this induction applies to the entire execution. From our induction hypothesis, it must be the case that, for all nodes n in t and the corresponding node n' in t', that their observable stores are identical. That is:

```
flattenStores t = flattenStores' [ ] t'                    (induction hypothesis)
        ⇒ obs (getEnv n) = obs' storeChain (getEnv n')
```

where storeChain is the result of recursing down the tree so far. Substituting for getEnv and the body of obs and obs' (defined above) gives us that for an arbitrary node n and its corresponding node n':

```
dirtyLocs » allLocs ⇔ writeStore » readStore » storeChain          (lemma)
```

Summary so far

For all thread trees t, t' which were created by program p and have evaluated an equal number of steps we assume:

```
flattenStores t = flattenStores' [ ] t'                    (induction hypothesis)
```

Therefore, for all nodes n(locs,dirty) in t and the corresponding node n'(readStore, writeStore) in t', the following must hold:

```
getOps n = getOps n'                                       --code of each node is identical
obs (getEnv n) = obs' storeChain (getEnv n')
⇒ dirtyLocs » allLocs ⇔ writeStore » readStore » storeChain     --substituting for getEnv, obs, obs'
```

Where storeChain is the accumulated environment above node n' in the tree t', as defined in flattenStores and identically in step'.

To continue, our original inductive step target is:

```
flattenStores (step t) = flattenStores' [ ] (step' t' [ ])          (induction step)
```

We will prove this using induction over storeChain, by proving for all corresponding n, n' that:

```
obs (getEnv (step n)) = obs' storeChain  (getEnv (step' n' storeChain))
```

We proceed as follows. The base case for the top-level node is:

```
obs (getEnv (step n)) = obs' [ ] (getEnv (step' n' [ ]))          (storeChain base)
```

Note the empty store chains on the right hand side. For this we can use:

```
obs (getEnv n) = obs' [] (getEnv n')
⇒ dirtyLocs » allLocs ⇔ writeStore » readStore » [ ]     --substituting for getEnv, obs, and obs'
⇒ dirtyLocs » allLocs ⇔ writeStore » readStore
```

In the last statement, the LHS and RHS are equivalent. This gives us another lemma we can use later in our case analysis. We must prove the inductive case:

```
obs (getEnv (step n)) = obs' storeChain (getEnv (step' n' storeChain))        (storeChain ind. case)
```

The induction hypothesis for this inner induction is that:

```
obs (getEnv (step x))
        = obs' storeChain (getEnv (step' x' storeChain))        (storeChain induction hypothesis)
```

holds for all x, x' nodes higher up the tree than n, n' so that the storeChain passed to the current node n, n' is correct. Therefore, the following holds for the current node:

```
obs (getEnv n) = obs' storeChain (getEnv n')
⇒ dirtyLocs » allLocs ⇔ writeStore » readStore » storeChain
```

We need to prove:

```
obs (getEnv (step n)) = obs' storeChain (getEnv (step' n' storeChain))        (storeChain ind. case)
```

We deal with the different case of step, step' on n, n'. Since we are performing induction over store chains, the LHS (naïve algorithm) is the same in the base case as with the induction case, since there is no storeChain. This simplifies the proof somewhat.

Summary of main proof

The induction is over store chains. In short, we have to show that:

```
obs(LHS) = obs'(RHS base)
obs(LHS) = obs'(RHS inductive)
```

for each algorithm construct in step, step'.

Our first case is the step of a Thread that has no operations:

```
LHS (naïve algorithm)
step (Thread env [ ]) = Thread env [ ]          --empty code sequences

RHS (optimised, base)
step' (Thread env [ ]) _ = Thread env [ ]

RHS (optimised, inductive)
step' (Thread env [ ]) _ = Thread env [ ]
```

All are identical, proving this simple case for n and n'. Now we deal with Threads that create a location.

```
--LHS (naïve algorithm)
step (Thread (s1,s2) (Create loc val : ops))
```

- {definition of step, substituting for case}

= Thread (envUpdate loc val env) ops

- {substituting for envUpdate}

= Thread (s1, storeUpdateStore loc val s2) ops

--RHS (optimised, base, inductive)

step' (Thread (readStore, writeStore) (Create loc val : ops) _

- {definition of step, substituting for case}

= Thread (envUpdate loc val (readStore, writeStore)) ops

- {substituting for envUpdate}

= Thread (s1, storeUpdateLoc loc val s2) ops

The base and inductive cases for step' here are identical, because the storeChain is not used. The LHS = RHS, proving the case. Now we come to the case where n and n' are Threads that perform a Move op. First the base case:

--LHS (naïve algorithm)

step (Thread (allLocs,dirtyLocs) (Move readLoc writeLoc:ops))

- {definition of step, subst. into case statement.}

= let val = envDeref readLoc (allLocs,dirtyLocs) in

Thread (envUpdate writeLoc val (allLocs,dirtyLocs) ops

- {defn of envDeref}

= Thread (envUpdate writeLoc (storeDeref readLoc **(dirtyLocs » allLocs)) (allLocs,dirtyLocs)**) ops

--RHS (optimised algorithm, base)

step' (Thread (readStore, writeStore) (Move readLoc writeLoc:ops)) [ ] --storeChain = [ ]

- {definition of step, subst into case}

= let val = envDeref readLoc (readStore » [ ], writeStore) in

Thread (envUpdate writelLoc val (readStore,writeStore) ops

- {remove [ ], subst val}

= Thread (envUpdate writeLoc (envDeref readLoc (readStore, writeStore)) (readStore,writeStore) ops

- {defn of envDeref}

= Thread (envUpdate writeLoc (storeDeref readLoc (**writeStore » readStore**))

(readStore,writeStore)) ops

Now the LHS and RHS are of the same form. There are two observables stores on the LHS and RHS. One pair is LHS (allLocs, dirtyLocs) and RHS (readStore,writeStore). By our induction hypothesis, these are observationally equivalent. The other observable store pair is (dirtyLocs » allLocs) on the LHS (naive algorithm) and (writeStore » readStore) on the RHS (optimised algorithm). This fits one of our lemmas:

dirty » locs ⇔ writeStore » readStore

Therefore the same result will be found by each storeDeref, and so updated by envUpdate, proving the base case. Now we deal with the inductive case for Move. Remember that for the naïve algorithm the base and inductive cases are the same, so we need only prove that LHS(naïve) = RHS(optimised, inductive).

```
--RHS (optimised, inductive)
step' (Thread (readStore,writeStore) (Move readLoc writeLoc:ops)) storeChain
```

- {definition of step', subst into case}

```
= let val = envDeref readLoc (readStore » storeChain, writeStore) in
        Thread (envUpdate writeLoc val (readStore,writeStore) ops
```

- {subst defn of envDeref}

```
= let val = storeDeref readLoc (writeStore » readStore » storeChain)
```

- {subst val}

```
= Thread  (envUpdate writeLoc (storeDeref loc (writeStore » readStore » storeChain))
        (readStore,writeStore) ops
```

This of the same form as the LHS, and the inner store expression fits our lemma:

dirtyLocs » allLocs ⇔ writeStore » readStore » storeChain

Thus, the same value is dereferenced in each case, proving our inductive case for move. Now we come to the case for stepping a thread that invokes a supervisor. The effects of InvokeSupervisor on a node spread to its newly invoked children, so we need to show that:

obs (getEnv (step n)) = obs' storeChain (getEnv (step' n' storeChain)) (storeChain ind. case)

for the parent node, and its children. However, proving this for one child node is sufficient since the children are created with a map, and are all the same (apart from their code, which is handled by the other proof cases).

```
--LHS (naïve algorithm)
step (Thread (allLocs,dirtyLocs) (InvokeSupervisor bodyCodeSeq threadCodeSeqs : ops)
```

- {defn of step, subst into case, subst body}

```
= let threads = [Thread (capture (allLocs,dirtyLocs)) codeSeq | codeSeq ← threadCodeSeqs] in
Supervisor (allLocs,dirtyLocs) (Thread (capture (allLocs,dirtyLocs)) bodyCodeSeq) threads ops
```

- {subst defn capture}

```
= let threads = [Thread (dirtyLocs » allLocs, [ ]) codeSeq | codeSeq ← threadsCodeSeqs] in
Supervisor (allLocs,dirtyLocs) (Thread (dirtyLocs » allLocs, [ ]) bodyCodeSeq) threads ops
```

```
--RHS (optimised, base)
step' (Thread (readStore, writeStore) (InvokeSupervisor bodyCodeSeq threadCodeSeqs : ops) [ ]   --storeChain [ ]
```

- {defn of step, subst into case, subst body}

```
let threads = [Thread ([ ],[ ]) threadCodeSeq | threadCodeSeq ← threadCodeSeqs] in
```

Supervisor **(readStore, writeStore)** (Thread ([ ],[ ]) bodyCodeSeq) threads ops

The LHS and RHS are of the same form, but with different environments for the supervisor, body, and threads. To prove this base case we must prove that the supervisor environments are equivalent, and that new body thread and child threads have observable system stores that correspondingly equivalent. Supervisor environments are unchanged from before step, so assuming the induction hypothesis hold then they are still observationally equivalent. That is, for the supervisor environment we have:

obs (allLocs,dirtyLocs) ⇔ obs' storeChain (readStore, writeStore)

- {subst for storeChain, obs, and obs'}

⇒ dirtyLocs » allLocs ⇔ writeStore » readStore » [ ]

- {removing [ ]}

⇒ dirtyLocs » allLocs ⇔ writeStore » readStore

This matches a lemma, proving that supervisor stores are equivalent. The new environments for the supervised threads and the supervisor body are the same. We can prove observational equivalence simultaneously. For threads c, c' we have:

obs (getEnv c) ⇔ obs' storeChain (getEnv c')

- {subst for getEnv and storeChain}

⇒ obs (dlocs » locs, [ ]) ⇔ obs' [ ] (getEnv n')

- {subst for obs and obs'}

⇒ dlocs » locs » [ ] ⇔ writeStore » readStore » [ ]

- {removing [ ]}

⇒ dlocs » locs ⇔ writeStore » readStore

This matches our lemma, and proves the base case for supervisor invocation. Now we deal with the inductive case.

--RHS (optimised, inductive)

step' (Thread (readStore, writeStore) (InvokeSupervisor bodyCodeSeq threadCodeSeqs : ops) **storeChain**

- {same as base case}

let threads = [Thread **([ ],[ ])** threadCodeSeq | threadCodeSeq ← threadCodeSeqs] in

Supervisor env (Thread ([ ],[ ]) bodyCodeSeq) threads ops

Again we must prove store equivalence:

obs (allLocs, dirtyLocs) ⇔ obs' storeChain (readStore, writeStore)

- {subst for obs, and obs'}

⇒ dirtyLocs » allLocs ⇔ writeStore » readStore » storeChain

This matches lemma, proving that supervisor stores are equivalent. For body/child threads c, c' we have:

obs (getEnv c) ⇔ obs' storeChain (getEnv c')

- {subst for getEnv}

⇒ obs (dlocs » locs, [ ]) ⇔ obs' storeChain (getEnv n')

- {subst for obs and obs'}

⇒ dlocs » locs » [ ] ⇔ writeStore » readStore » storeChain

This matches our lemma, and so proves the case for supervisor invocation. There are two more cases in step, step' Thread. These are Nop and Expose t. Nop is proven trivially, so we elide it, and Expose for Threads is not permitted, so we need not prove anything. The Focus compiler statically guarantees that threads cannot perform expose operations. This brings us to step, step' Supervisor, which has three cases. The first is the conversion from a Supervisor to Thread on completion of the body code:

```
--LHS (naïve algorithm)
step (Supervisor env (Thread bodyEnv []) _ cont) =
        Thread (expose bodyEnv env) cont
--RHS (optimised, base, inductive)
step' (Supervisor env (Thread bodyEnv []) _ cont) _ =
        Thread (expose bodyEnv env) cont
```

All are the same, completing proof for this case. This brings us to the two final cases for step, step' Supervisor. We attempt the simplest one first, where the supervisor body is not itself a supervisor, and does not perform an expose.

```
--LHS (naïve algorithm)
step (Supervisor (allLocs,dirtyLocs) body threads cont)
```

- {defn of step, subst into case}

```
= Supervisor (allLocs, dirtyLocs) (step body) [step thread | thread ← threads] cont


--RHS (optimised, base)
step' (Supervisor (readStore,writeStore) body threads cont) [ ]
```

- {defn of step, subst into case}

```
= let newStoreChain = writeStore » readStore » [ ] in
    Supervisor (readStore,writeStore) (step' body newStoreChain)
            [step' thread newStoreChain | thread ← threads] cont
```

- {subst newStoreChain, remove [ ]}

```
= Supervisor (readStore,writeStore) (step' body (writeStore » readStore))
            [step' thread (writeStore » readStore) | thread ← threads] cont
```

We have to prove that the two supervisor environments are equivalent. We omit this since it follows the same proof pattern we have already demonstrated. Equivalence of step, step' Supervisor then comes down to proving that:

obs (step body) = obs' (step' body (writeStore » readStore)) storeChain

obs (step thread) = obs' (step' thread (writeStore » readStore)) storeChain

We have already done this by cases (assuming we now go on to prove the final expose case), so this base case is complete. We omit the proof for the inductive case, since it follows a similar pattern to our previous proofs. Thus, we must now prove the final and most difficult inductive case, for step, step' Supervisor, where the body performs an expose.

```
--LHS (naïve algorithm)
step (Supervisor env (Thread bodyEnv (Expose t:ops)) threads cont)
    • {defn of step, subst into case}
    = let   (front,(thread:rear)) = splitAt (t-1) threads
            (locs,dlocs) = getEnv thread
            newEnv = expose (locs,dlocs) env
            newBody = Thread bodyEnv ops
            exposedThread = setEnv (capture newEnv) thread
            newThreads = (front ++ [exposedThread] ++ rear)
      in Supervisor newEnv newBody (map step newThreads) cont
    • {rewrite, subst for getEnv, expose, capture}
    = let   (front,(thread:rear)) = splitAt (t-1) threads
            newEnv = foldr (\(loc,val) acc → envUpdate loc val acc) env dlocs
            exposedThread = setEnv (snd newEnv » fst newEnv) thread
            newBody = (Thread bodyEnv ops) in
      Supervisor newEnv newBody (map step (front++[exposedThread]++rear)) cont
```

Now the RHS; we abbreviate pushStoreThread to pst, and originalValues to vals, for the purpose of brevity

```
--RHS (optimised, inductive)
step' (Supervisor env (Thread bodyEnv (Expose t:ops)) threads cont) storeChain
    • {defn of step', subst into case}
    = let   (readStore,writeStore) = env
        newStoreChain = writeStore » readStore » storeChain
        (front,(thread:rear)) = splitAt (t-1) threads
        (tRead,tWrite) = getEnv thread
        newEnv = expose (tRead,tWrite) env
        vals = storeDerefStore tWrite storeChain          --renamed to vals from originalValues
        newBody = pst vals (Thread bodyEnv ops)
```

```
    exposedThread = setEnv (capture newEnv) thread
    newThreads = (map (pst vals) front) ++ [exposedThread] ++ (map (pst vals) rear)
    in Supervisor newEnv newBody [step' thread newStoreChain | thread ← newThreads] cont
```

- {subst for expose, newStoreChain}

```
= let (readStore,writeStore) = env
    (front,(thread:rear)) = splitAt (t-1) threads
    newEnv = foldr (\(loc,val) acc → envUpdate loc val acc) env tWrite
    exposedThread = setEnv (snd newEnv » fst newEnv) thread
    vals = storeDerefStore tWrite storeChain
    newBody = pst vals (Thread bodyEnv ops)
    newThreads = (map (pst vals) front) ++ [exposedThread] ++ (map (pst vals) rear) in
        Supervisor newEnv newBody [step' thread (writeStore » readStore » storeChain)
                                    | thread ← newThreads] cont
```

- {subst newThreads}

```
= let   (readStore,writeStore) = env
        (front,(thread:rear)) = splitAt (t-1) threads
        newEnv = foldr (\(loc,val) acc → envUpdate loc val acc) env tWrite
        exposedThread = setEnv (snd newEnv » fst newEnv) thread
        newBody = pst vals (Thread bodyEnv ops)
        vals = storeDerefStore tWrite storeChain in
    Supervisor newEnv newBody [step' thread (writeStore»readStore»storeChain)
                            | thread ← (map (pst vals) front) ++ [exposedThread] ++ (map (pst vals) rear)] cont
```

This is in the same form as the LHS. We can prove that the observable store with respect to newEnv is equivalent in both the RHS and LHS. We can also prove that exposedThread is observationally equivalent in the RHS and LHS, and to step, step' exposedThread is the same by induction. That is, we have:

```
LHS: obs(newEnv) = RHS: obs'(newEnv) storeChain
LHS: obs(exposedThread) = RHS: obs'(exposedThread) storeChain
```

We must prove that the new body threads are observationally equivalent. Remember that the definition of pushStoreThread (abbreviated to pst) is:

```
pst locs thread = let
        (read, write) = getEnv thread in
                setEnv ((storeInsertStore locs read), write) thread
```

We have on the LHS:

```
newBody = Thread bodyEnv ops
```

- {rewriting}

```
= let (allLocs,dirtyLocs) = bodyEnv in
```

Thread (allLocs, dirtyLocs) ops

On the RHS:

newBody = pst vals (Thread bodyEnv ops)

- {subst defn pst and getEnv}

= let (read,write) = bodyEnv in

setEnv ((storeInsertStore vals read), write) (Thread bodyEnv ops)

- {defn of storeInsertStore, defn of setEnv}

= let (read,write) = bodyEnv in

Thread ((read » vals), write) ops

- {subst for vals}

= let (read,write) = bodyEnv in

Thread((read » (storeDerefStore tWrite storeChain)), write) ops

We must prove that:

obs (allLocs,dirtyLocs) = obs' ((read » (storeDerefStore tWrite storeChain)), write) storeChain

- {subst defn of obs, obs'}

dirtyLocs » allLocs ⇔ write » read » (storeDerefStore tWrite storeChain) » storeChain (*)

Remember that tWrite here is the write store of the thread being exposed. This gives us our intended result, because the store dereference of tWrite in storeChain will return a set of locations and values that *are already in* storeChain. Thus,

(storeDerefStore tWrite storeChain) » storeChain ⇔ storeChain

Rewriting * taking this into account leaves us to prove:

dirtyLocs » allLocs ⇔ write » read » storeChain

This fits our lemma as before, so we have proven the case. All that remains is observational equivalence for the threads that are not the exposed threads in the final Supervisor expressions above. It suffices to do this for one thread. We can now do this *before* the step, step' since we have proven equivalence for all other step cases. As a result, the proof for an individual thread is very similar to the proof for the supervisor body (which is not stepped), and so we omit it.

We have shown that our optimised algorithm implements the intended environment semantics in the Focus run-time system.

10: Conclusions

This thesis has addressed the issue of designing abstractions for use in programming Internet applications. In particular we are interested in those applications that demand prudent recovery from failure, independent of human interaction. We have shown that supervisors and persistent relative observables are useful programming abstractions for programming in the Web domain.

Its scientific contribution lies in three main areas, which we now discuss.

The Interpreted Exception

In Chapter 2, we presented the results of an extensive study into the failure and performance characteristics of the Web. Although similar studies exist, none are as broad, and in particular ours is the first to address the issue of dynamic rate across individual Web transfers and to present results relating to rate variability. Our experiments have shown us that the Web is a complex non-deterministic entity, which is amenable to a new approach of implementing failure models, more sophisticated than that of traditional methods involving timeout. By studying the behaviour of Web *observables* we gained insight into how they might be applied to programming Web applications.

From the experimental data we have examined, we find that the majority of failures that occur on the Web are the result of timeout. Timeout accounts for more than three-quarters of all failures. Timeouts are *interpreted* failures based on observation of the properties of a Web transfer, in this case observation of the time that the transfer is taking. The nature of the corresponding underlying failures cannot be determined, since they are not absolute.

The behaviour of Web observables gives information that may be used in interpreting failure based on the notion of *acceptability*. Setting a timeout amounts to enforcing a *constraint* on the maximum time observable allowed for a particular operation. However, use of timeout alone is inflexible, and can be unreliable and inefficient. There are several observables in addition to time that can be used in setting the bounds of what is considered acceptable performance. Failure of the performance of a Web fetch to fall within these bounds generates an *interpreted exception*. We see it as self evident that the more information that is available, the more confident one can be that any interpretation of failure based on that information is appropriate.

Human browsers do not employ concrete notions of absolute failure to the Web. Certainly, they react to absolute failure when it is encountered, but our experiments and those of Zeus technologies indicate that interpreted failure is more important since it 'occurs' more

frequently. By examining the methods of failure interpretation employed by human browsers, in conjunction with our quantitative experiments, we identified the observables of transfer rate, connection latency, transfer time, and completion percentage. Furthermore, we notice that humans interpret failure in terms of complex relationships between constraints on the values of observables. An HCI study might have revealed the essence of this, but we have instead opted for as flexible an approach as possible since imposing unnecessary limitations on a programming system by design is imprudent. For example, the failure semantics employed by an automated agent, mobile in the Internet, are unlikely to be exactly the same as those of a human browser performing a similar task. As a result, we wish to allow a full range of expression in implementing failure semantics, rather than attempt to model specifically those failure semantics characteristic in human browsing behaviour.

In Chapter 3 we presented a programming methodology designed to maximise flexibility in interpreting failure. This methodology is based on the use of higher-order functions, but also has an analogous implementation in object oriented programming systems. In short, the technique allows arbitrary program logic specifying constraints on observables and the relationships between those constraints to be passed as a parameter to the Web fetch abstraction. Since this logic can cause the raising of exceptions, this allows a degree of parameterisation for flow control after failure. However, exception handling is a serialised model that does not integrate well with concurrency. Concurrency is extremely important in Internet computation, since it allows the CPU to be utilised during the periods of high I/O overhead incurred by Internet access. This motivates us to seek a new programming model that exposes domain properties while cleanly integrating concurrency, flexible failure interpretation, and flow of control for failure. The cradle of our model is the concept of a historical context for the observables of Web fetches.

The Essence of Internet Computing

Experienced Web users interpret failure based on something more than just the immediately available observables. Humans use available observables in conjunction with an expectation of the likely future behaviour for a particular Web fetch or series of Web fetches. This expectation is based on previous experience with that server, URL, country in which the server resides, and experience of Web failure and performance in general. In short, when interpreting failure, historical context is key. In Chapter 2, we show that under similar network and sever load conditions the performance of a particular Web site is consistent. Deviations in observable performance characteristics from those experienced in the past provide an invaluable clue when interpreting failure, or more accurately, when interpreting performance that is unacceptable.

In Chapter 5, we introduced a conceptual domain for Web programming based on the persistent relative observable. A persistent relative observable differs from a normal observable in that it is calculated relative to a historical context associated with either a particular URL or server. Persistent relative observables allow programmers to express computational logic in terms that are similar to the human failure model for the Web. However, they are not entirely designed to allow the modelling of human patterns of failure interpretation. Persistent relative observables have four very important benefits in their own right:

- *Portability* – since persistent relative observables are calculated as a ratio, they are independent of absolute units of measurement. This means that a program can be executed in any network context without modification, and be expected to exhibit similar failure semantics. For example, a program statement that dictates the interpretation of failure when rate falls below 20% of the historical average has the same semantics when executed on systems with T1 connectivity and when executed on a system with only a modem connection.
- *Mobility* – by similar arguments to the benefit of portability, applications can be mobile in the network without having to employ reflection techniques or complicated program logic that directs program flow to code appropriate to the network context. Mobile agents can migrate at will, without regard to the connectivity of their target host.
- *Future proofing* – programs that contain statements such as "interpret failure on transfer rate less than 5Kb/sec" make a concrete assumption about the speed of the local network connection, and of the Internet in general. The performance of the Internet is improving over time, since perhaps somewhat surprisingly the investment in infrastructure is outstripping the explosion in usage. Since overall performance is improving, the notion of what is acceptable performance is likely to become more constrained. Thus, statements like those above are likely to be quickly invalidated by the general trend towards increased bandwidth in the entire Internet. However, with our mechanism gradual changes in Internet performance are absorbed by the persistence mechanism, rendering all programs future proof.
- *Generalisable to all computation* – since persistent relative observables are independent of absolute units of measurement, they can be applied in situations that under normal circumstances would be meaningless. Persistent relative observables allow the interpretation of failure by the notion of acceptability based on the norm. Deterministic computations have predictable failure and performance characteristics, so we can

associate 'normal' observables with them. For example, the unit of measurement Kb/sec cannot be meaningfully applied to string processing operations. However, given that the rate of a deterministic string computation can be considered 'normal' at all times, we can meaningfully associate a persistent relative rate observable of 1.0.

Portability, mobility, and future proofing for programs are significant benefits. However, generalisation to all computation is perhaps the most important, because it is fundamental to the design of our high-level Web programming abstraction, the *supervisor*.

Concept Integration

In Chapter 3, we present three design goals for a Web programming system. Primarily, these are concerned with cleanly and orthogonally integrating the concepts of concurrency, flexible failure interpretation, and flow control for failure, while bringing the properties of the Web domain into the semantic space. Although we do not claim that these design goals are the most appropriate for all Web programming models, we feel that languages or systems based on them are likely to allow the expression of more concise and intuitive programs than with more general purpose programming languages.

Supervisors are essentially concurrency constructors, which linguistically and semantically separate control logic in the supervisor body from computational logic in the parameter threads. They monitor and control the behaviour of concurrent computations, but in a general way that is independent of the particular computation taking place. Supervisors are founded on the persistent relative observables mechanism, and all computations export the set of Web observables. Since persistent relative observables are generalised to all computation, supervised computations may consist of an arbitrary sequence of Web fetches and deterministic computations, the nature of which the logic of the supervisor need not be concerned with. Supervisors are general control functions that are useful for specifying generalised failure semantics for a class of computations. The persistent relative observables mechanism maps the observable behaviour of those computations into a uniform pattern so that computations with very different performance characteristics can be controlled by the same supervisor.

In supervisors, we have developed a programming language abstraction that exposes the properties of the Web domain, while cleanly integrating the means for concurrency, flexible failure interpretation, and flow control for failure. Focus is the first programming language to integrate these concepts. Furthermore, by presenting examples we have shown that supervisors can concisely express many useful patterns of computation, including some that

map directly onto human 'algorithms' for Web computation. Since programming is essentially about mapping the solution to a problem from human thought into something that a computer can execute, the conciseness and intuitiveness of our example programs suggest that supervisors are an appropriate abstraction for programming in the Web domain.

The supervisor is a high level abstraction, in particular with respect to its mechanism for automatic backwards error recovery. However, we show in Chapter 9 that although the environment mechanism logically duplicates the entire store on each thread inception, this behaviour can be modelled efficiently. In the same chapter, we show that supervisors conceptually contain the service combinator algebra.

Further Work

Focus is intended for applications that are affected by the failure and performance characteristics of the Internet. Examples of these are Web crawlers, distributed applications, and mobile agent systems. To gain insight into the usefulness of particular language features and gain insight into how we might refine the language, it is important to deploy such applications in the target domain. However, to deploy these applications, it is necessary to have control of several Web servers. This itself is not a problem, but because Focus is intended to cope with the failure and performance properties of the Web, several co-located severs do not provide an accurate reflection of the 'real' computing environment. To test the mechanisms that deal with performance and failure we need control of a diversity of servers, particularly in geography and loading. Currently, we do not have the resources for this, but in the future we hope to be able to deploy a limited set of diverse servers and implement a broad class of applications over them.

Although our performance experiments provided some useful results, larger scale experiments are necessary. We did not control any of the target servers in our experiments, and although this heterogeneity and autonomy is intrinsic to the Web, it makes it more difficult to attribute particular results to specific circumstances. For example, with several different sites world wide, we could simulate different levels of load on each of them under controlled conditions. This would allow us to distinguish between performance and failure characteristics that arise from server load and those that arise from network load. Network load is consistent according to time of day, but server load is largely unpredictable. If we can distinguish performance and failure patterns that arise from server load and network load, then this might be useful in determining appropriate action. For example, if a server is found to be under heavy load, this would encourage the seeking of alternate resources rather than retrial. If an area of the network is under heavy load, retrial at a later time is an option, but it

might be sensible to prioritise further Web fetches from servers that are in a different geographical location.

Studies in comparative programming might show that task implementations in Focus are more concise, elegant, and intuitive that those written in a traditional GP language such as Java. This thesis has contained very little comparative programming. However, we do not claim that our Web programming model is *better* than that employed by users of Libwww or the Java Web API. Instead, we state simply that our model is appropriate for the class of applications that we are interested in, and that in it we have discovered some useful new concepts and how to integrate them cleanly with existing concepts. Certainly, proving that Focus is better than Java, say, would be a significant result. However, such a result is unlikely to change anything. For this reason, and the fact that comparative programming is lengthy and detracts from the presentation of concepts, we have chosen not to include it in this thesis.

Opinionated Final Words

Focus is an experimental language and we cannot realistically expect it to be deployed to any level by the Web programming community at large. Moreover, evolving and maintaining an industry strength programming language is an enormous task that we do not wish to undertake. However, the design of an experimental programming language is not about trying to encourage people to use that language. Developing a domain specific programming language from scratch allows researchers to concentrate on important and relevant concepts, ignoring problems that might arise from issues such as compatibility with existing infrastructure. Only large software companies and standards organisations have the clout to change the way the world uses the Web. However, they tend to become involved in an endless cycle of compromises in order to maintain backward compatibility. For example, in the 1990's, Microsoft dominated the operating systems market. However, it is widely believed that the progression from MS-DOS to Windows 3.1 to Windows 95, was in technical terms a disaster. Had there been the will to move to carefully designed operating systems and programming languages that broke from the past, we might already be free of the long running 'software crisis'. Primarily for economic and business-political reasons, this has not happened.

Perhaps more relevant to us is the Web itself, which is an example of what happens when we allow physicists to play with computers. Ever since its rise in popularity, the Web has been plagued with half-adopted proprietary extensions, largely unimplemented portions of server protocols, and compromise extensions intended to patch up the original mistakes. Had the Web been designed properly from the outset, it is even possible that this thesis would not have been necessary.

We have designed a domain specific language for programming the Web, and we do not expect people to use it. However, in designing Focus without being distracted by industry hype, we have produced a clean and simple programming model that captures what we believe to be the essence of Internet computing. It is our hope that this contribution to knowledge will have pragmatic implications for the designers of Web applications. For example, we hope that Web application programmers will abandon the sole use of timeout in favour of the better model that incorporates all forms of observables. And in particular, we hope that persistent relative observables are adopted by Web APIs, in some form or another.

Appendix – The Focus Language

Focus is a simple language intended primarily as a vehicle for the supervisor construct. In this appendix, we provide a summary of the language, and details of its implementation.

Focus Syntax

Following is a formal syntax definition for Focus, in Backus Normal Form (BNF). The focus grammar is LL(1).

Program ::= *seq* | *e*

seq ::= *dec* [; *seq*] |
E [; *seq*]

dec ::= **let** *I* = *E* | *Bind identifier to value*
type *I* **is** *T* | *Bind identifier to type*
forward *I* **is** *T* | *Forward declaration of identifier as type*

E ::= *E0* |
supervisor (*label* : *T* [, *label* : *T*]* → *T*) **is** *E* | *Supervisor value*
function (*label* : *T* [, *label* : *T*]* → *T*) **is** *E* | *Function value*
[[*I* = *E*] [, *I* = *E*]*] | *Structure values*
loc(*E* **)** | *Location value*
file(*E* **)** | *File value*
set([*E*] [, *E*]* **)** | *Set value*
vector (*E*..*E* ← *E*) | *Vector value*
not *E* | *Logical negation*
- *E* | *Arith negation*
sleep *E* |
card *E* | *Set cardinality*
take *E* | *Set extraction*
waitfor *E* | *Conditional sleep*

		get *E*		*| Web get*
		post *E* ← *E*		*| Web post*
		foreach *E* **do** *E*		*| Set iterator*
		while *E* **do** *E*		|
		observerop I		|
		controlop I		
E0	::=	*E1* [**:=** *E*]		*Assignment, lhs must be location*
E1	::=	*E2* [**or** *E2*]*		
E2	::=	*E3* [**and** *E3*]*		
E3	::=	*E4* [*relop E4*]		
E4	::=	*E5* [*addop E5*]*		
E5	::=	*E6* [*mulop E6*]*		
E6	::=	*E7* [*setop E7*]*		
E7	::=	*E8* **(** [*E*] [**,** *E*]* **)**	|	*Function or supervisor application*
		E8 [**.** *label*]*		*Structure dereference*
E9	::=	*E10* [# *E*]*		*Vector indexing*
E10	::=	**at** *E*	|	*Location dereference*
		if *E* **then** *E* **else** *E*	|	
		if *E* **do** *E*	|	
		(*E* **)**	|	
		fail	|	

I |

literal |

{}

observerop ::= **rate** | **latency** | **time** | **completion** | **prob** |

active | **suspended** | **failed** | **done**

controlop ::= **suspend** | **activate** | **retry** | **expose**

relop ::= < | > | = | != | <= | >=

addop ::= + | - | ++ *Last is string concatenation*

mulop ::= **mul** | **div** | **mod**

setop ::= **union** | **intersect** | **difference**

literal ::= **true** |

false |

numeral |

string |

T ::= **void** | *No type*

bool |

int |

string |

mime |

loc(*T* **)** |

set(*T* **)** |

vector(*T* **)** |

supervisor(*T* **)** |

function([*T*] [, *T*] → *T* **)** |

	{ [*I* : *T*] [, *I* : *T*] }	|	*Structure type*
	I		*Type identifier*

I ::= standard identifier not a keyword and not containing special characters

Lexical rule – in *seq*, a new line after *E* of type void will be read as a semi-colon symbol, if the next symbol is not a closing brace.

Focus Concepts

The Focus language model is imperative with block structure and all values are first class (including functions and supervisors). It is expression based in that there is no restriction on the application of language constructs within expressions. Focus is strongly and statically typed, with type equivalence being defined structurally. However, Focus allows the naming of types. The semantics of equivalence is defined by value equality for scalar types and as identity for all others. Identity is defined as existing between two values if they were both originally constructed with respect to the same binding association (let statement). Therefore, identity cannot exist between anonymous values. Focus is a garbage collected language, and the allocation and deallocation of physical memory is entirely automated.

One slightly unusual feature of Focus is that it provides an explicit location type with the following syntax (from BNF above):

E	::=	**loc(** *E* **)**	*(Location value)*
E0	::=	*E2* [**:=** *E*]	*(Assignment)*
E10	::=	**at** *E* | ...	*(Dereference)*

The Focus location model requires all declarations of mutable values to be of type *location*. Any other binding is immutable. This encompasses structure fields, function parameters, vector and set elements, and is defined recursively, across all types. In the examples below, note the different meanings of the assignment operator ":=", and the binding association operator "=".

```
let a = 0
let b = loc(0)
a:= 42          //Illegal, a is an integer
b:= 42          //OK, b is an integer location
b:= b + 1       //Illegal, adding an integer to a location
b:= at b + 1    //OK, adding an integer and an integer and assigning into a location
let c = at c
c:= 42          //Illegal, c is an integer
let d = b
d:= 42          //OK, d is a location (sharing identity with b, and updates reflect
this)
```

It is important to understand the concept of *bindings* and *identifiers* here. The identifier a is bound to the integer value *0,* and so all uses of the identifier a result in this value. However, the identifier b is bound to an integer *location.* Therefore, uses of b result in the location, and *not* the value contained therein. In order to obtain the contents of a location, the keyword *at* must be used as a dereference operator.

Focus Compiler

The Focus compiler is classic recursive descent [], and the syntax analysis aspect of it maps directly from the BNF definition. Lexical analysis, syntax analysis, and code generation all take place within a single pass of program source. There is no persistent byte code representation for Focus programs. Instead, every time a program is run, it is compiled into a sequence of abstract machine instructions on the fly, which are then executed. This 'sequence' of instructions is actually a graph of instruction objects, each of which support an *execute* operation and a *nextInstruction* operation. At run-time one or more threads traverse this graph.

Environments

As compilation proceeds, a *static environment chain* is maintained that reflects the namespace of the current lexical scope. A static environment is essentially a set of mappings from identifiers to integers, where the integers are indices into an array. The idea behind environments is to compute where declared values are to be stored in memory. As identifiers are declared, space is reserved for them in a dynamic environment array, and they are given a unique index within that array. When an identifier is used in program source, the static environment chain is searched recursively, and if the identifier is found the number of levels of scope out and the index within that environment is known. During run time, when a new

level of static scope is entered, a *dynamic* environment object of the appropriate size is created and associated with the thread. This new dynamic environment is associated with a reference to the previous dynamic environment, in much the same way as the static environment. The 'previous' environment is that for the immediately enclosing level of lexical scope. In turn, this environment references the next outermost level of scope, and so on. When a level of dynamic scope is exited, the current dynamic environment is discarded, and replaced with the one that it references, the next outer scope level.

A major difference between static (compilation) environments and dynamic (run time) environments is there is no identifier or type information available in dynamic environments. Instead, during compilation instructions are generated that are parameterised with the scope and index values calculated from the static environment chain. These values are used to iterate through the dynamic environment chain and index the appropriate field in the dynamic environment (essentially an untyped array). In short then, all environment access requires an index pair, a scope offset and value offset, which are both calculated statically. However, environment assignments are always performed to the current environment, so only require the value offset and not the scope offset.

Dynamic environments are created unpopulated. However, in Focus it is not possible to declare an identifier without binding it to a value. This means that environment fields are statically guaranteed to be valid (non-empty) when they are first referenced. Note that *environments are not mutable*, they simply store values. Now in Focus, locations are themselves values, but location update does not change the location, only it's contents. In the run-time system, locations are references to heap objects, just like all other non-scalar values. However, *LocationHeapObjects* are mutable.

Given the previous point, we can now state that environments are independent of the read and write stores intended to control concurrent update. Read and write stores capture the actions of location read and update. It is the contents of locations that are shadowed in the read and write stores. Essentially, the read and write stores present a mapping from the location value (a reference, which in the implementation is an integer), to the contents of that location *with respect to the current thread.* Threads need not shadow the contents of environments since they are not mutable and independent of whatever level of concurrency is present. On a related note, complex cyclic data structures are unaffected by concurrency for the same reason – they are always immutable. However, any locations within them will be shadowed correctly by the read and write store mechanism.

Functions, Supervisors, and Environments

In Focus, functions and supervisors are first class, meaning that they may be assigned to locations in outer scope levels and be passed as parameters to other functions or supervisors. This means that the environment of a function or supervisor declaration may be completely different from that of the invoking context. However, correct semantics for function and supervisor invocation demands that functions and supervisors must be invoked with the environment context of their declaration. This means that they must carry what is called their 'closure' through assignment and parameter passing. A closure object is an instruction reference (code pointer) – the function/supervisor entry point, and an environment reference – the start of the environment chain for that function. When a thread calls a function or invokes a supervisor, the current environment is replaced with the function environment from its closure. The environment of the calling context is stored on the stack, to be popped off and restored when the function or supervisor returns.

Garbage Collection

All run time objects in Focus derive from a *HeapObject* class which has an abstract method, *mark*, and a Boolean, *marked*. Every subclass of *HeapObject* must implement a *mark* method. The idea is that invoking *mark* results in the particular object setting its *marked* flag, and recursively invoking *mark* over anything reachable from it. For example, *CompositeHeapObjects*, which represent structure objects at run time, recursively invoke mark over all their non-scalar components. Locations, threads, and even instructions are all represented as HeapObjects. Threads recursively mark their stacks, current instruction, and read and write stores. To garbage collect the entire system, simply invoke *mark* on the thread that is the root of the thread tree.

Focus has scalar values as well as reference values. Internally these are treated as machine words. Since some heap objects may reference other values, for example structures, vectors, and locations, they need to be able to determine whether the particular value is a reference or a scalar. We achieve this with reference maps and flags. For example, when a structure object is created it is passed an array of Booleans that associates with an internal array of values that constitute the structure fields. By mapping this pointer flag array, the structure heap object knows which of the fields are to be converted to *HeapObject* references and which are scalars to be ignored. The same mechanism is used for run-time system stacks, where a reference flag stack is grown and shrunk in parallel with the actual stack. For run-time objects such as locations and vectors, we need only construct them with a reference flag as opposed to a map.

Threading and Asynchrony

Instruction sequences are represented as directed cyclic graphs. Branches in the graph are caused by conditional instructions (choice) and concurrency (n-split). The execution of a concurrency instruction causes the creation of a new lightweight thread. The Focus run time system consists of a tree structure of these threads, each consisting of a stack, environment, code pointer and read and write stores. 'Executing' a thread causes it to execute its current instruction and request the next (advance the code pointer), then executes any child threads it may have. In this way, executing the top-level thread causes the whole thread tree to be recursively executed for one *cycle*. Program execution ends when the top-level thread executes a *Halt* instruction.

Usually, instruction execution takes negligible time. However, asynchronous operations such as Web fetch must be executed asynchronously. *Asynch* heap objects keep track of ongoing asynchronous operations. When an asynchronous operation is executed, the run-time system creates an appropriate *Asynch* object and stores it in the thread object, then the thread scheduler moves on to the next thread, without advancing to the next instruction. The next time this thread is scheduled, the *Asynch* object is found and interrogated as to it's status. If it is completed or failed, then it is destroyed and the thread is advanced to the next instruction as normal. Since *Asynch* objects are heap objects, they are garbage collected along with all other heap objects. Different implementations of *Asynch* (derived classes) are expected to implement correct handling of resource release upon their collection.

Bibliography

[1] The Hyper Text Transfer Protocol.
http://www.w3.org/Protocols/

[2] W3C Hypertext Markup Page.
http://www.w3.org/MarkUp/

[3] Uniform Resource Locators.
http://www.w3.org/Addressing/URL/

[4] The Common Gateway Interface.
http://www.w3.org/CGI/

[5] S. D. Reilly et al – Increasing the Computational Potential of the World Wide Web.
Technical report CS-96-02, University of Virginia.
ftp://ftp.cs.virginia.edu/pub/techreports/CS-96-02.ps.Z

[6] Netscape Communications – Persistent client state, http cookies.
Preliminary specification, 1996.
http://home.netscape.com/newsref/std/cookie_spec.html

[7] A. K. Weissinger – ASP in a Nutshell
O'Reilly, February 1999, ISBN: 1565924908

[8] S. McPherson – Java Server Pages: A Developers Perspective.
http://developer.java.sun.com/developer/technicalArticles/Programming/jsp/index.html

[9] C. Musciano – How to Get Started with Server-side Includes.
Netscape World, January 1997.
http://www.netscapeworld.com/netscapeworld/nw-01-1997/nw-01-ssi.html

[10] The Altavista Web search engine.
http://www.altavista.com/

[11] Amazon Web bookstore.
http://www.amazon.com/

[12] Search Spaniel search engine.
http://www.searchspaniel.com/

[13] ECMA-262 JavaScript Language Specification.
Netscape Corporation white paper.
http://developer.netscape.com/docs/javascript/e262-pdf.pdf

[14] J. Gosling and H. McGilton – The Java language environment.
Sun Microsystems white paper, May 1996.
ftp://ftp.javasoft.com/docs/papers/langenviron-pdf.zip

[15] J. Hunter and W. Crawford – Java Servlet Programming.
O'Reilly, December 1998, ISBN: 156592391X

[16] D. Chappell – Understanding ActiveX and OLE.
Microsoft Press, August 1996, ISBN: 1572312165

[17] S. Williams and C. Kindel – The Component Object Model.

Microsoft MSDN online, 1994.
http://msdn.microsoft.com/library/default.asp?URL=/library/techart/msdn_comppr.htm

[18] Dorte Toft – Netscape's Browser Share Now 25 Percent
PC World News, July 2000
http://www.pcworld.com/pcwtoday/article/0,1510,12203,00.html

[19] Mobile Code
http://www.w3.org/MobileCode/

[20] J. Vitek and C. Tschudin (eds.) – Mobile object systems: towards the programmable Internet.
Lecture Notes in Computer Science, vol. 1419, 1998.

[21] Luca Cardelli – Global computation.
ACM Computing Surveys 28(4es), December 1996,
http://www.luca.demon.co.uk/Bibliography.html

[22] Luca Cardelli – Abstractions for Mobile Computation.
Lecture Notes in Computer Science, Vol. 1603, 1999.
http://www.luca.demon.co.uk/Bibliography.html

[23] J. Vitek, C. Tschudin (eds.) – Mobile Object Systems: Towards the Programmable Internet.
Lecture Notes in Computer Science Vol. 1222, Springer, 1998.

[24] P. Buneman – Semi-Structured Data.
16th ACM Symposium on Principles of Database Systems. ACM Press 1997, pg. 117-121.

[25] D. Suciu – An Overview of Semi-Structured Data
SIGACT News , vol. 29 , no. 4 , pp. 28-38 , December 1999.

[26] Extensible Markup Language (XML)
http://www.w3.org/XML/

[27] Document Object Model (DOM)
http://www.w3.org/DOM/

[28] P. Mockapetris – Domain Names: Concepts and Facilities.
Network Working Group, RFC 1034.
http://www.faqs.org/rfcs/rfc1034.html

[29] Zeus Technologies Web Performance Survey
http://webperf.net/

[30] F. Cristian – Understanding Fault-Tolerant Distributed Systems.
Communications of the ACM, Vol. 34(2), Feb 1991, pp56-78.

[31] L. Cardelli – A Language with Distributed Scope.
Computing Systems, 8(1):27-59, January 1995

[32] D.E. Bakken and R.D. Schlichting – Supporting fault-tolerant parallel programming in Linda.
IEEE transactions on parallel and distributed systems, 6(3), March 1995.

[33] The OMG – Common Object Request Broker: Architecture and Specification.
Object Management Group Document Number 91.12.1, 1991.

[34] A. Black et al – Distribution and Abstract Types in Emerald.
IEEE Transactions on Software Engineering, SE-13(1), Jan 1987.

[35] J. Waldo et al – A Note on Distributed Computing.

Sun Microsystems Technical Report, SMLI TR-94-29, Nov 1994.

[36] A. Wollrath and J. Waldo – RMI Tutorial Trail.
http://java.sun.com/docs/books/tutorial/rmi/TOC.html

[37] G. R. Andrews and R. A. Olsson – The Evolution of the SR Language.
Distributed Computing 1(2), April 1986.

[38] D. May – Occam Language
ACM SIGPLAN Notices, 18(4), pp69-79, April 1983.

[39] M.D. Byrne et al – The Tangled Web we Wove: a Taskonomy of WWW Use.
Proc. CHI'99, pp544-551, Addison Wesley, 1999.

[40] L.D. Catledge and J.E. Pitkow – Characterizing Browsing Strategies in the WWW.
Proc. International World-Wide Web Conference, 1995 (WWW5).
http://www.igd.fhg.de/archive/1995_www95/papers/80/userpatterns/
/UserPatterns.Paper4.formatted.html

[41] Java Standard Edition Platform Documentation.
http://java.sun.com/docs/

[42] Zeus Technologies Ltd
http://www.zeus.com/

[43] Solving the World Wide Wait, Zeus Technologies Web Performance Site
http://webperf.net/

[44] Transmission Control Protocol – RFC 793
http://www.faqs.org/rfcs/rfc793.html

[45] Bob Metcalfe - From the Ether,
Column in Infoworld Magazine, Dec. 4, 1995, page 61

[46] D. Nicholas and P. Williams – Journalism and the Internet, The Changing Information Environment, Pre-fieldwork literature review.
http://www.soi.city.ac.uk/~pw/ji_lit.html

[47] John S. Quarterman – Imminent Death of the Internet?
http://www.mids.org/mn/606/death.html

[48] The Joint Academic Network (Janet)
http://www.ja.net/

[49] NTL
http://www.ntl.com/

[50] Keith Sibson – Measuring Web Performance
http://hippo.cs.strath.ac.uk/WebPerformance/

[51] Netcraft Web Server Survey
http://www.netcraft.co.uk/survey/

[52] iPlanet E-Commerce Solutions (Server previously Netscape Enterprise Server)
http://www.iplanet.com/

[53] R. Levin – Program structures for exceptional condition handling.
Ph.D. Thesis, Carnegie-Mellon University, June 1977.

[54] D.W. Flater et al – Some extensions to the C language for enhanced fault detection.
Software Practice and Experience, 23(6), pp617-628, June 1993.

[55] N. H. Gehani – Exceptional C or C with exceptions.
Software Practice and Experience, 22(10), pp827-848, October 1992.

[56] L. Cardelli and R. Davies – Service Combinators for Web Computing.
IEEE Transactions on Software Engineering, Vol 25, No 3, May-June 1999. pp 309-316.
http://www.luca.demon.co.uk/Bibliography.html

[57] H. Marais – WebL, a programming language for the Web.
In Computer Networks and ISDN Systems (WWW7), 30, pp259-270, April 1998.
http://www.research.digital.com/SRC/personal/Johannes_Marais/pub/www7/

[58] Keith Sibson – Adding Persistent Relative Observables to the Service Combinator Algebra.
http://hippo.cs.strath.ac.uk/ServiceCombinators/

[59] Libwww: The W3C Protocol Library
http://www.w3.org/Library/

[60] The Oz2 Programming Language
http://www.ps.uni-sb.de/oz2/

[61] A. Romanovsky – Extending conventional languages by concurrent exception resolution.
Journal of Systems Architecture, v. 46, No. 1, pp.79-95. 2000 (author sent preprint).

[62] L. Cardelli – Wide Area Computation.
Lecture Notes in Computer Science, Vol. 1644, Springer, 1999. pp. 10-24
http://www.luca.demon.co.uk/Bibliography.html

[63] B. Randell et al – Reliability issues in computing system design.
ACM Computing surveys, 10(2), June 1978.

[64] L. Cardelli and P. Wegner – On understanding types, data abstraction, and polymorphism.
Computing Surveys, 17(4):471-522, 1985
http://www.luca.demon.co.uk/Bibliography.html

[65] M.P Atkinson and R. Morrison – Persistent First Class Procedures are Enough.
Lecture Notes in Computer Science 181, pp223-240. 1984.

[66] B. Randell et al. – Coordinated Atomic Actions: from Concept to Implementation.
Technical Report, Computing Dept., University of Newcastle upon Tyne, TR 595, 1997
http://www.cs.ncl.ac.uk/research/trs/papers/595.ps

[67] R. Connor and K. Sibson – HCL: A Language for Internet Data Acquisition.
Proceedings of Workshop on Internet Programming Languages, ICCL'98.
http://hippo.cs.strath.ac.uk/papers/hcl.ps

[68] B. Randell and J. Xu – The evolution of the recovery block concept.
Software Fault Tolerance, Lyu ed. Wiley 1995.

[69] J. J. Horning et al - A Program Structure for Error Detection and Recovery.
Operating Systems, LNCS, 172-187, Springer-Verlag, 1974.

[70] System structure for software fault tolerance – B. Randell
IEEE transactions on software engineering, 1(2), June 1975.

[71] B. Randell et al – Fault tolerance.
In Predictably dependable computing systems, ESPRIT basic research series.

[72] B. Randell et al – Reliability issues in computing system design.
ACM Computing surveys, 10(2), June 1978.

[73] Free Online Dictionary of Computing
http://foldoc.doc.ic.ac.uk/

[74] Arnold Pear – Notes on Two Phase Locking and Commit Protocols
http://www.csd.uu.se/~arnoldp/distrib/TwoPhase.html

[75] Oracle Corporation
http://www.oracle.com/

[76] S.W. Loke and A Davison – Logic Programming with the World Wide Web.
Proc. 7th ACM Conference on Hypertext 1996, pp235 – 245, March 1996.
(paper) http://www.cs.mu.oz.au/~swloke/papers/TR-95.33.ps.gz
(figures) http://www.cs.mu.oz.au/~swloke/papers/figs1.ps.gz

[77] S.W. Loke – Adding Logic Programming Behaviour to the World Wide Web.
Ph.D. Thesis, University of Melbourne, August 1998.
http://www.cs.mu.oz.au/~swloke/logicweb-thesis.html

[78] L. Sterling and E. Shapiro – The Art of Prolog.
2nd Edition, MIT Press, 1994.

[79] A. Davidon and S.W. Loke – A Concurrent Logic Programming model of the Web.
University of Melbourne technical report 98/23. November 1998
http://www.cs.mu.oz.au/~swloke/papers/conmod.ps.gz

[80] E. Shapiro – The Family of Concurrent Logic Programming Languages.
ACM Computing Surveys Vol. 21(3), Sept 1989, pp413-510.

[81] NCSA Mosaic Common Client Interface
http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/CCI/cci-spec.html

[82] K.M. Kavi ed. – Real-Time Systems, Abstractions, Languages, and Design Methodologies.
IEEE Computer Society Press, 1992.

[83] A.D. Stoyenko – A Schedulability Analyzer for Real Time Euclid.
Proc. IEEE 1987 Real-Time Systems Symposium, pp218-225, 1987.

[84] P.I.P. Boulton – A Process Control Language.
IEEE Transactions on Computers, C-18(11), 1969, pp1049-1053 (also in [82]).

[85] A.D. Stoyenko – The Evolution and State-of-the-Art of Real-Time Languages.
Journal of Systems and Software, pp 61-84, April 1992.

[86] K.B. Kenny and K.J. Lin – Building Flexible Real-Time Systems using the FLEX Language.
Computer, 24(5), pp70-78, May 1991.

[87] K.J. Lin and S.M. Natarajan – Expressing and Maintaining Timing Constraints in FLEX.
Proc. IEEE Symposium Real-Time Systems Symposium, 1988, pp96-105, (also in [82]).

[88] K.M. Kavi and S.M. Yang – Real-Time Euclid: A Language for Reliable Real-Time Systems.
Journal of Systems and Software, pp85-99, April 1992.

[89] J.B. Goodenough – Exception handling design issues.
ACM SIGPLAN Notices, 10(7), July 1975.

[90] F. Cristian – Exception Handling and Tolerance of Software Faults.
Software Fault Tolerance, Lyu ed. Wiley 1995.

[91] M. D. MacLaren – Exception handling in PL/I.
Proc. ACM Conference on Language Design for Reliable Software, March 1977.

[92] Wasserman – Procedure-oriented exception handling.
Technical Report 27, UCLA.

[93] J.B. Goodenough – Exception handling: issues and a proposed notation.
Communications of the ACM, 18(12), pp683-696, December 1975.

[94] J.J. Horning – Programming languages for reliable computing systems.
Lecture Notes in Computer Science, 69, pp494-530, 1978.

[95] S. Yemini and D. M. Berry – A modular, verifiable exception handling mechanism.
ACM Transactions on Programming Languages and Systems, 7(2), pp214-143, April 1985.

[96] A. Wijngaarden et al – Revised report on the algorithmic language Algol 68.
Acta Informatica, 5, 1975.

[97] B. Liskov – CLU Reference Manual.
Lecture Notes in Computer Science Vol. 114, Springer, 1981.

[98] T. Anderson and P. A. Lee – Fault Tolerance: Principles and Practice.
Prentice Hall, 1981.

[99] M. D. MacLaren – Exception handling in PL/I.
Proc. ACM Conference on Language Design for Reliable Software, March 1977.

[100] R.E. Sweet – The Mesa programming environment.
SIGPLAN Symposium on Language Issues in Programming Environments, ACM, 1985.

[101] M. Gauthier – Exception handling in Ada-94 (*sic*).
ACM SIGADA Ada Letters, 15(1), pp70-82, 1995.

[102] J.J. Horning – Effects of programming languages on reliability.
In Computing Systems Reliability, Cambridge University Press, 1979.

[103] A. Koenig and B. Stroustrup – Exception handling for C++.
Journal of Object-Oriented Programming, pp16-33, July 1990.

[104] B. Venners – Exceptions in Java.
Java World, 3(7), July 1998.
http://www.javaworld.com/javaworld/jw-07-1998/jw-07-exceptions.html

[105] P. M. Melliar-Smith and B. Randell – The role of programmed exception handling.
Proceedings of an ACM Conference on Language Design for Reliable Software, ACM, 1977.

[106] B. Liskov and A. Snyder – Exception handling in Clu.
IEEE Transactions on Software Engineering, 5(6), pp546-558, November 1979.

[107] N. Cocco and S. Dulli – A mechanism for exception handling and its verification rules.
Journal of Computer Languages, 7, 1982, pp89-102.

[108] H. Custer – Inside Windows NT.
Microsoft Press, 1993.

[109] L. R. Nackman et al – AML/X: a programming language for design and manufacturing.
IBM Research Report No. RC11992.

[110] B. Liskov et al – Abstraction mechanisms in Clu.
Communications of the ACM, 20, pp564-576, August 1977.

[111] J.L. Knudsen – Exception handling, a static approach.
Software Practice and Experience, 14(5), pp429-449, May 1984.

[112] J.L. Knudsen – Better exception handling in block structured systems.
IEEE Software, May 1987.

[113] R. D. Tennent – Language design methods based on semantic principles.
Acta Informatica, 8, pp97-112, 1977.

[114] R. Bird – Introduction to Functional Programming using Haskell.
2nd edition, Prentice Hall, 1998.

[115] S. Thompson – Formulating Haskell.
University of Kent Technical Report 29-92, 1993.
http://www.cs.ukc.ac.uk/pubs/1992/123/index.html

[116] J. O'Donnell and G. Rünger – A formal derivation of a parallel binary addition circuit.
Computing Science Department TR-1995-19, University of Glasgow (1995).
http://www.dcs.gla.ac.uk/~jtod/publications/adder-TR95-19.ps.gz