Logic Programming as a Formalism for Specification and Implementation of Computer Systems

by

Anthony Joseph Kusalik

B.Sc., The University of Lethbridge, 1978
M.Sc., The University of British Columbia, 1982

A thesis submitted in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

in

The Faculty of Graduate Studies
Department of Computer Science

We accept this thesis as conforming
to the required standard

The University of British Columbia

June 27, 1988

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date June 27/88

Abstract

The expressive power of logic-programming languages allows utilization of conventional constructs in development of computer systems based on logic programming. However, logic-programming languages have many novel features and capabilities. This thesis investigates how advantage can be taken of these features in the development of a logic-based computer system. It demonstrates that innovative approaches to software, hardware, and computer system design and implementation are feasible in a logic-programming context and often preferable to adaptation of conventional ones. The investigation centers on three main ideas: executable specification, declarative I/O, and implementation through transformation and meta-interpretation. A particular class of languages supporting parallel computation, committed-choice logic-programming languages, are emphasized. One member of this class, Concurrent Prolog, serves as the machine, specification, and implementation language.

The investigation has several facets. Hardware, software, and overall system models for a logic-based computer are determined and examined. The models are described by logic programs. The computer system is represented as a goal for resolution. The clauses involved in the subsequent reduction steps constitute its specification. The same clauses also describe the manner in which the computer system is initiated. Frameworks are given for developing models of peripheral devices whose actions and interactions can be declaratively expressed. Interactions do not rely on side-effects or destructive assignment, and are term-based. A methodology is presented for realizing (prototypic) implementations from device specifications. The methodology is based on source-to-source transformation and meta-interpretation. A magnetic disk memory is used as a representative example, resulting in an innovative approach to secondary storage in a logic-programming environment. Building on these accomplishments, a file system for a logic-based computer system is developed. The file system follows a simple model and supports term-based, declarative I/O. Throughout the thesis, features of the logic-programming paradigm are demonstrated and exploited. Interesting and innovative concepts established include: device processes and device processors; restartable and perpetual devices and systems; peripheral devices modelled as function computations or independent logical (inference) systems; unique, compact representations of terms; lazy term expansion; files systems as perpetual processes maintaining local states; and term- and unification-based file abstractions. Logic programs are the sole formalism for specifications and implementations.

Table of Contents

List of Figures

Acknowledgement

Much credit is due to Harvey Abramson, Carl McCrosky, Keith Clark, Ian Foster, Graem Ringwood, Michael Hirsch, and Ehud Shapiro for their support, criticisms, and contributions. The patience and assistance of my professors at the University of British Columbia and colleagues at the University of Saskatchewan are gratefully acknowledged. Harvey Abramson, David Etherington, and Carl McCrosky provided valuable comments on earlier drafts of portions of this thesis.

This thesis would not have been possible without the loving support, patience, and understanding of my wife and family. This thesis is dedicated to them.

1. Introduction

The field of logic programming, though still very young, has attracted the interest of many in Computer Science. I.C.O.T. and their Fifth Generation Computer Project [Moto82] have intensified and focused interest in the area [McCo83]. Logic programming assumes an atypical model of computation in which

- logic is used to express information in a computer,
- logic is used to present problems to a computer, and
- logical inference is used to solve the problems.

Logic programming has recognized advantages for various types of non-numeric applications. Hence there is a worldwide effort in the development of new logic-programming languages, software, support hardware, and applications.

The expressive power of logic programs permits the application of conventional computer system constructs to logic-based computers. However, not all traditional concepts are well accommodated in a "logical context". Also, the properties of logic-programming languages allow novel techniques and models to be employed. This thesis demonstrates that innovative approaches to software, hardware design, and computer system models are feasible in the logic-programming context, and often preferable to adaptations of conventional approaches. In general, the questions dealt with in this thesis are: how can a logic-based computer system and its components be specified and implemented in a logic-programming language? What are the results of doing so? Can interactions with peripheral devices be described declaratively (that is, solely through the declarative constructs of logic programs)? Finally, how can such declarative interaction be realized for a nontrivial device and significant computer service (e.g., file storage)? A significant feature of the work is that logic, as realized by committed-choice logic-programming languages, is used as the sole formalism and mechanism for specifying and implementing components of a computer system. The logic-programming language used is Concurrent Prolog [Shap83a, Shap86b, Shap87]. It is assumed that the reader is familiar with the rudiments of logic programming. The thesis incorporates material that has appeared before [FoKu86, Kusa85b, Kusa86, Kusa87], as well as new work and results.

This thesis does not focus on computer systems in general, but on logic-based systems specifically. The techniques and mechanisms have not been thoroughly investigated for other types of computer

systems. Some types of systems may be difficult to deal with, especially those employing incompatible constructs (such as interrupts).

The goals of this thesis are given in more detail in Section 1.3. Section 1.2 provides motivation and background. The methodology for achieving the goals is the subject of Section 1.6. Section 1.4 describes the (computer) languages used and general system characteristics assumed. The use of logic-programming languages for this type of research is justified in Section 1.5. We begin with an explanation of necessary terminology.

1.1. Terminology

Logic programming is the direct use of logic as a programming language. That is, a **logic-programming language** is a machine-intelligible form of symbolic logic. A "program" consists of a set of axioms (a theory), and a "computation" is the construction of a proof of a consequence of the program [Kowa74]. A **logic-inference machine** is a computer which realizes this model of computation.

A **parallel logic-programming language** is one which supports the direct expression of concurrent or parallel computations or activities. **Committed-choice logic-programming languages** are a class of these languages which utilize the "guarded command" concept [Dijk76] to control OR-parallelism. They display "don't-care nondeterminism" [ClGr81]. Thus, if multiple clauses exist for resolution of a goal, only one is used. It is chosen based on the success of the subgoals preceding the **commit** construct in each of the candidate clauses.

A **logic-based operating system** is an operating system for a logic-inference machine. It is implemented in a logic-programming language. An inference machine and a compatible logic-based operating system constitute a **logic-based computer system.**

Throughout this document, a "logical term" – or sometimes just "term" – refers to a term in a logical language. A **term** is a variable, constant symbol, or expression of the form

$$f(t_1, \ldots, t_j)$$

where f is a j-place function symbol and $t_1, \ldots, t_j$ are terms. A term that contains no uninstantiated variables is a **ground term.**

1.2. Background and Motivation

Development of highly parallel "conventional" computers has met with only moderate success, due mainly to difficulties in distributing a computation over a substantial number of processors and synchronizing their operation. This has led to the argument that, in order to exploit even modest parallelism, new models of computation must be developed. Logic programming is recognized as one possible approach [Bic84]. Logic programs possess four potential forms of parallelism [CoKi81]:

- OR-parallelism – the simultaneous trial of program clauses whose heads are all unifiable with a goal.
- AND-parallelism – the concurrent resolution of each unit goal of a clause body.
- Stream parallelism – the concurrent, coordinated processing of structured data.
- Search parallelism – the parallel search of a distributed program database for candidate clauses.

Not surprisingly, various languages, computational models, and machine architectures supporting parallel computation through logic programs have been proposed. Their relative strengths are still very much in debate.

The development of logic-programmed computer systems is a goal of some intensive research efforts [Marc84, McCo83, Moto82]. However, it is an excursion into areas where little practical experience exists. The evolution of "new generation" computers requires novel approaches to computer architectures, languages, software designs, and applications [Moto82]. Hence, studies which contribute useful information and results in these areas are needed and worthwhile. Work is progressing on the theoretical aspects of logic programming [JaLM86], on the design of abstract hardware models [TaAS87], development and implementation of languages [FGRS86, HoSh86, TaFu86, TaSS87], and application of these languages [BLMO86, KHKH87, Shap87].

An area receiving much less attention, but of no less importance, is the design, modelling, and implementation of operating systems for inference machines [Fost87b, Shap83c, SHHS86, TYUK84]. Given the power of logic-programming languages, it is possible to adapt conventional software and system designs for logic-based computers. This was done, for example, with ICOT's prototypic logic-based operating system, SIMPOS [HaYo83, TYUK84]. Translating software into a logic-programming language can often yield a better understanding of its design, and even reveal (implicit) inconsistencies [Kowa82]. However, rather than translations and adaptations, a fresh, top-down, declarative approach

can result in new insights into operating systems, a better understanding of underlying concepts, and innovative ideas or approaches. Such fresh approaches have not been extensively pursued. Few declarative treatments of basic concepts such as computation or "task" control, input/output, initialization ("bootstrapping"), and exception handling have been developed. Such investigations may provide valuable input to language and inference machine design and development. Studies of logic-based operating systems are not premature. Despite the unavailability of target hardware, they are possible assuming abstract machines with reasonable characteristics.

A language must be used extensively to ascertain its relative merits. The language used here, Concurrent Prolog, is no exception. Papers have appeared demonstrating its utility in various applications [Shap87]. However, many methods of achieving parallel execution in logic programs have been proposed and are being studied [Bowe82, ClGr81, EiKM82, EmLu82, FuKM82, GoTM84, Naka84, Pere82, Ueda86a, YaAi86]. The qualities and constructs of Concurrent Prolog are often debated. Thus, tests of Concurrent Prolog's expressive power and computational model are (still) in order. Justifications for extensions or restrictions of the language, or its dialects, are also valuable.

The feasibility of systems programming in parallel logic languages has been demonstrated [ClGr84b, Shap83c]. Succinct specifications for traditional operating system structures such as command interpreters, device drivers, and queue managers were presented. It was shown that representative, nontrivial aspects of an operating system can be handled cleanly using the languages. However, a declarative approach was not always followed. This suggests that the potential benefits of the using logic-based languages for systems programming have not been fully realized.

It is not uncommon to see formal logic used for specification and verification of computer components. Temporal logic is a popular choice [Abad86]. Prolog has also been employed in this capacity [BCMD87, UeKa83], as have parallel logic-programming languages [Suzu86, WeSh86]. However, the specifications are often of lower-level hardware operation. Much less frequently they are of software or high-level hardware characteristics. The potential benefits from pursuing these latter possibilities warrant their investigation.

Even when logic programs are used to specify major software components, the power of the approach is often not fully exploited. For example, Shapiro [Shap83c] gives a high-level specification of

an operating system with a "reboot" capability by way of a concise Concurrent Prolog program. Implications for hardware characteristics, however, are not examined, nor is an integrated operating system model presented. Hence, more work in this area is needed.

A frequently-cited advantage of logic-programming languages is their capability for executable specification: the axioms (program statements) which describe a model also implement the model [Kowa79]. Thus, a single logic-programming language can be used for both specification and implementation. Although this feature of logic programs is often cited in the literature, it is much less frequently exploited. It is rarely demonstrated for large, complex programs or substantial components of a computer system. For example, presentations of ICOT's prototype inference machine, PSI [UYYT83, YYTN83], and its operating system, SIMPOS [HaYo83, TYUK84], make little use of executable specifications. This thesis explores and exploits this capability.

In a software engineering context, program verification establishes that an implementation meets its specification [Broo87]. Usually, specification and implementation languages are different. Cumbersome, complex transformations are necessary for automatic generation of programs from specifications. Furthermore, developing a complete and consistent specification is a major task; debugging is difficult. Using logic programs for specifications greatly aids in these tasks. Since the specification is in a representation which can be manipulated by computer, checks for consistency can be automated. The dual nature of logic programs means that the specification is an implementation [Kowa79], transformations are unnecessary, and verification is much simplified. These advantages and capabilities of logic programs for specification are known, though not commonly exercised. More attention should be drawn to them.

Logic programs can be manipulated by source-to-source transformations. Because of logic programming's strong formal basis, these transformations are simple, yet powerful [TaSa83]. They have been applied to various tasks with promising results. For example, transformations have been used to make programs more efficient [SeFu87, Ueda86b, Ueda87], to implement language extensions [Bloc84, CoSh86, HiSS86], to aid in program development [SaTa84], and in partial evaluation [Take86, Vase86]. Even more applications and techniques are foreseen, making research into source-to-source transformation of logic programs desirable.

A conventional computer consists of a processor, a memory, and various peripheral devices. Input/output is achieved by modification of state of the various devices. Conventional programming languages reflect this notion of I/O through destructive assignment and side-effects. Hence it is not surprising that I/O operations within common logic-programming environments are achieved by side-effects. This is contrary to the basis of logic programming, however. Logic-programming languages treat computation as controlled deduction, not as destructive assignment to memory locations. Further, side-effects greatly complicate the task of software verification. In the context of parallelism or concurrency, side-effects render the overall behaviour of a system almost unpredictable [EiKM82]. An alternative to side-effecting operations for I/O is necessary.

In a conventional system, interaction with peripheral devices typically entails interrupts. However, it is not at all clear how a mechanism such as interrupts can be captured by logic programs. Occurrence of an exception has no meaning in the underlying formal logic [Fost87b]. It has been proposed that the cleanest way to achieve communication with peripheral devices in logic-based computers is to have devices consume or generate Concurrent Prolog streams [Shap83c]. Yet in many cases the treatment of I/O reverts to questionable constructs at some lower level. For example, the following clause has been suggested as part of the specification in Concurrent Prolog of a terminal keyboard device [Shap83c]:

instream([X/Xs]) :- read(X) | instream(Xs).

The goal *read(X)* is the guard of the clause. The basis for guards in Concurrent Prolog is Dijstra's guarded commands. Guarded commands check the "state" of a computation to determine whether to proceed with a particular statement. In the example above, the guard does not check or verify a state, but effects one. Clearly, there is need for improvement in this area.

Within a logic-programming environment the ability to construct and manipulate terms is natural: terms are the fundamental (and only) method for representing data. Arguments in structured terms must (recursively) be terms. Goals and clauses can also be treated as terms, as shown by Bowen and Kowalski [BoKo82]. It seems only proper, then, to have I/O within logic programs, and within logic-inference machines, based on terms. Yet the more usual approach is a character-oriented interface to external resources. Logic variables are not supported, and the interface requires changes in representation resulting in inefficiencies. Thus, term-based I/O requires investigation.

Logic-based computer systems require nonvolatile secondary storage or file systems, just as conventional computer systems do. However, in most existing logic-programming environments files, nonvolatile storage, and other external resources are accessed and manipulated by nondeclarative commands. These operations make use of side-effects and destructive assignment. For example, Prolog implementations typically support access to files through "system predicates" similar in style and functionality to file operations in procedural languages. (Sometimes they even have the same name!) Quintus Prolog™ [Quin87], for instance, includes the following system predicates for dealing with files and I/O:

get(Char)	*flush_output(Stream)*	*stream_position(Stream, Old, New)*
put(Char)	*open(File, Mode, Stream)*	

Prototype research systems are sometimes little better. For instance, Logix [SHHS86], a user environment for the Flat Concurrent Prolog emulator, provides basic file utilities which are nothing more than simple interfaces to the file I/O facilities of the underlying UNIX™ operating system. Predicates supported include *cd/2* to change the working directory, *get_file/4* to read information, and *put_file/4* to write. In such instances, file system access may be efficient, but lacks the benefits of declarative programming. Investigation of declarative interfaces to secondary storage and files is clearly necessary. The feasibility of this alternate approach has been demonstrated by declarative treatments of input such as "query-the-user" [Serg83]. Also, some Prolog implementations support "more logical" file access predicates [McCl88]. Development of workable declarative secondary storage models and file systems for logic-based computer systems are expedited by many of the capabilities and characteristics of logic-programming languages already mentioned.

The development of a logic-based computer system or its components need not abandon all traditional concepts and techniques. Existing, proven concepts may still be utilized. Reformulation is necessary if conventional approaches are awkward in a "logical" context, or an alternative is superior. Also, new approaches motivated by the logic-programming paradigm may even be applicable to conventional architectures.

In summary, progress has been made towards the development of logic-based computer systems. However, much remains to be done. Application of the unique properties of logic programs to computer hardware and software is a worthy and timely portion of that work.

1.3. Objectives and General Methodology

The general goal of this thesis is an answer to the question: how can better use be made of the novel properties of logic programs in the design, specification, and implementation of logic-based computer systems? An answer is pursued by investigation of three main ideas: executable specification, declarative I/O[1], and implementation through transformation and meta-interpretation. A variety of issues regarding inference machines, logic programming, systems software, and operating systems are involved. This section details individual subgoals for satisfying the main goal. The subgoals are all interrelated.

Specific objectives of the thesis are as follows.

- The first goal is to determinate hardware and software models for a logic-based computer system. The chosen models must be integrable into a single, cohesive, overall computer system model. All models must utilize features of the logic-programming paradigm. The hardware model should be driven by software and logic-based operating system considerations, yet be reasonable in light of current or prospective technological capabilities. Peripheral devices must be covered by the hardware model, and the models must allow a declarative treatment of interaction with peripheral devices. Interesting characteristics of the models resulting from the logic-programming paradigm are to be examined.
- The thesis' second aim is to demonstrate and exploit the power of logic programs for executable specification. The hardware, software, and overall computer system models mentioned above are to be specified. The fact that a single logic program serves for both specification and implementation will be stressed.
- Given that a logic-based computer system can be specified, a subsequent goal is to investigate how the system might be initiated ("bootstrapped").
- As will be shown, a single logic program can serve for both specification and implementation of computer components. For software components, an implementation is thus readily available. However, for hardware components, such as peripheral devices, the specification program may not be a

1. "Declarative" is used here in the sense of "declarative languages", especially as to how they differ from imperative or procedural ones.

practical implementation. This presents a difficult problem. It also provides a fourth objective: to explore a general, systematic, and practical method for deriving an alternate, tailored implementation corresponding to the specification program. The objective is to demonstrate the use of meta-interpreters and source-to-source transformations to obtain implementations from specifications, or to obtain more efficient implementations, in the context of logic programming.

- Another primary goal is to extend the declarative model of computation offered by logical languages to peripheral devices, such as secondary storage media. The principle of term-based, declarative I/O is to be demonstrated, and its feasibility shown. Devices whose actions and interactions are free of nondeclarative constructs such as interrupts, side-effects, and destructive assignment are sought. Terms are to be the basic units of I/O, and unification the primary mechanism for data transfer. Models following these ideals are to be devised and specified. An implementation, in the form of a working prototype of a "declarative I/O device", is to be realized.
- Given the development of a declarative I/O device, it is natural to pursue a nontrivial "higher-level" computer system component – incorporating both hardware and software – which also supports declarative, term-based I/O. File storage is a natural example. Following the general goal for the thesis, a unique and simple view of file storage is sought which eliminates many of the complications normally associated with storage systems. Executable specifications and novel properties of logic-programming languages are to be used in specifying a file system model, and in achieving a working implementation. Novel aspects of the resultant file system are to be outlined.

Throughout this work, unique and interesting properities of logic-programming languages, especially committed-choice ones, are to be exploited. Innovative and beneficial results of doing so will be highlighted and explained.

A related, though separate, goal of the thesis is to demonstrate the expressive power of logic-programming languages. The ease with which computations, especially parallel ones, can be described is to be shown. This work is to be a further test of the expressive power and computational model of the chosen committed-choice logic-programming language, Concurrent Prolog, and its dialects.

Many prospective languages, computational models, and machine architectures for logic-based computer systems are currently being investigated. For this reason, results of more general applicability

are preferentially pursued. Emphasis is necessarily placed on conceptual, rather than empirical, results.

1.4. Language and Target System

In this thesis, only logic-programming languages founded on Horn-clause logic are considered. (The logic-programming language Prolog is a member of this group.) Languages can be, and have been, based on other forms of logic [Colm86, GoMe86, JaLa87, MaMW86, SuYo86]. However, those based on Horn-clause logic typically lend themselves to more efficient implementation and have been more extensively investigated.

Concurrent Prolog is the primary language of this work. It is assumed as the specification, implementation, and underlying machine language. A dialect, Flat Concurrent Prolog [Mier84, MTSL85], is used for "practical" implementations. Concurrent Prolog was chosen because of its expressivity, and because implementations of the language and its dialects are available. However, the results presented are also applicable to other parallel logic-programming languages. The abbreviations "CP" and "FCP" are used for Concurrent Prolog and Flat Concurrent Prolog, respectively, throughout this work. A description of Concurrent Prolog is given in Section 2.1, along with justification of its choice.

To take advantage of parallelism inherent in logic programs and to achieve high execution speeds, proposed inference machine architectures are typically highly parallel, multiprocessor configurations. For example, ICOT's ultimate inference machine is envisaged as a parallel logic-inference engine consisting of hundreds of processing units, a structured memory, and a communication network [Uchi83]. Unless otherwise stated, in this document the term "logic-inference machine" is assumed to refer to a parallel architecture (a uniprocessor would be a degenerate case).

1.5. Why Work With Logic-Programming Languages?

Several characteristics of logic programming make it an interesting research tool. The dual declarative/operational nature of logic-programming languages allows the same statements to both describe the knowledge necessary to solve a problem, and to implement a solution. Logic-programming languages are very high-level, so complex ideas and computations can be expressed concisely. For example, systems programs in Concurrent Prolog are more succinct than those written in more conventional languages [Shap83c]. As well, algorithms and models are easily implemented and later modified. Logic programming reconciles the requirements that a programming language be natural and easy to

use, yet be machine intelligible. Logic-programming languages can express parallelism and concurrent computation clearly and easily. They have a cleaner semantics than procedural languages, which facilitates analysis and proofs of correctness. It has been observed that translation of software from a procedural to a logic-programming language can often yield a better understanding of the software's design, and even reveal implicit inconsistencies [Kowa82]. In addition, novel solutions to existing problems are often possible.

The use of logic-programming languages benefits their development and acceptance. The interdisciplinary aspects of this thesis should help make their noteworthy capabilities more widely known. Also, as stated earlier, experience contributing to comparative analysis of the varied forms of logic-programming languages, especially those supporting parallelism, is valuable.

There exist specification languages which are more descriptive than Concurrent Prolog. There are also specification languages which have more efficient implementations. Automated transformation techniques are known for mapping from "descriptive" languages to "efficient" ones. Concurrent Prolog is a good compromise between these two levels of specification language. It is descriptive and implementable with moderate efficiency. And no inter-level transformations are necessary. Thus, use of Concurrent Prolog for specifications is acceptable.

1.6. Detailed Methodology and Synopsis of Content

The detailed methodology for achieving the outlined goals is now presented. The discussion also provides a synopsis of the thesis.

The objectives of this thesis are pursued mainly in Chapters 3 through 6. These chapters presume certain foundational information, which is provided in Chapter 2. There, the committed-choice logic-programming language selected for the work is described, and its selection is justified. General hardware characteristics and the architectural model assumed for logic-based computer systems are given. A design plan for logic-based operating systems is also provided. A key concept in later discussions, "device process", is introduced.

In Chapter 3, logic programs, as expressed in Concurrent Prolog, are used to specify a computer system model. Hardware and software components are captured, as well as the interfaces between them. Development of the logic-based computer system model proceeds in a top-down manner, as in

resolution of the goal *computer_system*. Clauses to solve this goal provide a high-level specification of the machine and operating system. The principle of executable specification figures prominently in the discussion. The clauses of a high-level specification of the operating system, for example, form part of its implementation. It is demonstrated that through the properties of committed-choice logic-programming languages, systems and hardware components with novel characteristics can be developed. Thus "restartable" and "perpetual" systems and components arise and are explored. Various representations of hardware errors are examined. The concepts involved are concisely captured in CP programs. The dual operational/declarative nature of logic programs allows the computer system specification to also describe the manner in which initialization takes place. Thus, an initialization ("bootstrapping") mechanism is investigated. Executable specifications mean that the highest-level clauses in the operating system specification are the first computations of it performed at initialization. Taking advantage of executable specification is more difficult in the case of hardware components. While the capability permits a correct (software) simulation of a device, a working implementation is more difficult. It is possible, however, as is shown in Chapter 5.

Peripheral devices are central to the remaining chapters. For continuity and brevity, a single, representative hardware component is selected to serve as an illustrative example. Generality is not lost if a commonplace and moderately complex device is chosen. A nonvolatile, high-capacity, direct-access secondary storage device (a magnetic disk) is therefore used. This choice is further justified in Chapter 4. The use of a secondary storage device yields an auxiliary result: the remainder of the thesis constitutes a treatment of "declarative secondary storage".

The stage is now set for Chapter 4 which explores declarative I/O: interaction that can be described without destructive assignment or side-effects between application or system software and peripheral devices. Logic programs are known to be appropriate for the task [Shap83c, Somo87]. Models of peripheral devices supporting declarative I/O are developed. The models are consistent with the computer system specifications presented in Chapter 3. Data transfer is in units of terms. Since a secondary storage medium is used as a representative device, operations involve information storage and retrieval. Two general techniques for developing the models are expounded. The first views a computer system as a collection of independent inference systems. Stored information is a set of facts – a knowledge base. The technique is powerful, though variables shared between inference systems pose a significant

potential problem. Two solutions to this problem are examined in the context of disk storage. One involves renaming of variables; the other forces all stored information to be ground terms. The first development technique also allows an extension to the storage models for reclamation of space. The other technique for developing device models views activities of peripheral devices as computation of special functions. Storage device models are again formulated and examined. It is shown that, using either technique, finite and infinite storage can be accommodated. The applicability of the techniques to other peripheral devices is demonstrated. A noteworthy concept that arises in the discussion is "unique, compact representation". Its utility is shown in Chapter 6. Device models are specified in Concurrent Prolog. The specification programs are thus executable, though they may not directly constitute an effective implementation.

Given models for peripheral devices which support declarative, term-based I/O, it is natural to seek a practical or prototype conforming implementation. Again using secondary storage as a representative example, Chapter 5 examines a method for doing this. The method consists of transformations and an application of meta-interpretation. One of the storage models developed in Chapter 4 is chosen. The specification program in Concurrent Prolog is subjected to a sequence of limited extensions and equivalence-preserving transformations. The result is an enhanced model whose Concurrent Prolog specification program can be translated to Flat Concurrent Prolog. The resultant FCP program is amenable to a reformulation which moves the modification of device content from object level to meta-level [BoKo82]. An enhanced meta-interpreter supports the reformulated specification program. Analysis of the program constructs manipulating the device content show that the enhanced meta-interpreter can be directly implemented. The functionality of the constructs is provided by supplementary term-based I/O primitives. These can be understood declaratively, but are simple enough to be implemented directly. The result is a prototype implementation which corresponds to the original model. The feasibility of a working, "production" device based on this prototype is also considered. The implementation method described is general, being applicable to other devices, other types of (device) models, and other languages (not just CP and FCP).

Chapter 6 once again extends and builds on preceding work. A file system is developed which utilizes the prototypic secondary storage facilities. This provides an opportunity to demonstrate and test the capabilities of the secondary storage device model formulated earlier, and of its implementation. It

also presents another chance to investigate the concept of term-based, declarative I/O. Initially, a simple and basic file system model is developed and succinctly formulated as a logic program. Inessential details, which only complicate file systems, are avoided. The file system devised includes a "file system kernel" that maintains a set of associations between files and filenames. Basic file system services – creation, access, and removal of files – are provided by the kernel. Information transfer is unification-based, so file abstractions do not require explicit read and write constructs. "File system servers" extend the functionality of the kernel and provided enhanced services. Servers are de-emphasized in the discussion; concentrations is on the file system kernel. Various extensions to the file system model are considered. The specification of the file system kernel, with only minor modification, is shown to form an implementation. Stable storage of files is based upon checkpointing of the kernel's local "data state", the "file system database". The information is recorded by the declarative secondary storage device developed previously. Both the model and (naive) implementation are critically analyzed. Using the knowledge thus gained, a more refined model is formulated, specified, and discussed. Through several enhancements, this new model corrects deficiencies in the original model and provides greater functionality. Yet it retains the same basic features and character. As before, the file system kernel is emphasized in the discussion. The kernel now maintains a "file system history". An implementation is realized easily and described. A number of further enhancements to the file system model are still possible and practical. As a final step, one such enhancement – support of hierarchical directories – is pursued. Even this last file system is strongly based on the original model. For example, file operations can be understood declaratively, information is transferred through unification, and stable storage is achieved through checkpointing. The "unique, compact representation" concept introduced in Chapter 4 makes the latter practical. The development of a file system in a logic-programming context yields several interesting concepts, including file system histories and "lazy term expansion". These are highlighted.

At this point, the goals of the thesis have been satisfied. Use of the novel properties of logic programs in the design, specification, and implementation of computers systems and their components has been explored. The principles of executable specification, declarative I/O, and implementation through transformation and meta-interpretation have been dealt with extensively. Therefore, Chapter 7 provides a summary and conclusion, reiterating important aspects of the thesis.

Related works are considered in relevant context within individual chapters.

Each of the main chapters is, for the most part, self-contained; each is designed to be comprehensible with only minor reference to surrounding chapters. The chapters all rely heavily on background information supplied in Chapter 2, however. Therefore, readers of selected portions of the thesis are encouraged to begin with Chapter 2.

2. Setting The Stage

To proceed with this work, it is necessary to choose a language, describe the architectural model for the system, and outline the design for the operating system. Such is the subject of this chapter. Motivation and justification for the choice of language is given.

2.1. Language

The best-known logic-programming language, Prolog [Rous75, Warr77], is a poor candidate as a single specification, systems programming, and machine language: it does not allow the expression of concurrent computations without resort to side-effects. A variety of languages supporting concurrency have been described, some being "extended" forms of Prolog [ClMG82, Hogg82, Naka82, Naka84, Wise82] and others based more directly on the underlying computational model [ClGr81, ClGr84a, EmLu82, PeNa84, Ueda86a]. A member of the latter group, Concurrent Prolog [Shap83a, Shap86b, Shap87], is used here. Concurrent Prolog, hereafter denoted "CP", facilitates the expression of concurrency, communication, synchronization, and indeterminacy, yet is very concise. CP is a descendant of the Relational Language of Clark and Gregory [ClGr81]. Many effective programming constructs and techniques, such as objects, class hierarchies, (unbounded- and bounded-buffer) stream and channel communications, message-passing, eager- and lazy-enumeration of solutions, wave-front computation, and encapsulation can be cleanly realized using the language [HiCF84, KTMB86, MBTL87, Shap83a, Shap84b, ShTa83, TaFu83, TMKB87]. CP has also been employed in a wide variety of applications, from parsing to knowledge-programming [EdSh84, FuTK83, HeSh84, Hira83, Kusa84a, ShMi84, ShSh83, Suzu86, WeSh86]. On the whole, results favor its use for expression of high-level or complex concepts [Shap86a]. Experimental implementations of the language have been realized [LeGo85, MiTC85, Shap83a, UeCh85]. A prospective architecture for a CP machine has also appeared [Shap83b, Shap84b]. Thus CP is chosen as the machine, implementation, and specification language for this thesis.

The following table summarizes the manner in which CP embodies familiar computational concepts [Shap83a]:

Concept	Concurrent Prolog Construct
Process	Unit goal
System	Conjunctive goal
Process state	Value of arguments
Process computation	Goal reduction
Process communication	Unification of shared variables
Process synchronization	Suspending unification of read-only variables

A unit goal corresponds to a short-lived process. Tail-recursive goal satisfaction corresponds to a **persistent** or **perpetual** process [Warr82] (the sequence of subgoal invocations is viewed as a single, longer-lived computational entity). This **process interpretation** is key to its usefulness for systems programming.

CP machines do not (yet) exist. Even implementing a "practical" CP machine emulator is currently technically difficult [Levy84]. However, practical machine simulators are possible for more restricted languages, such as Flat Concurrent Prolog [HoSh86, Mier84, MTSL85] or a related language, PARLOG [ClGr84a, ClGr86, FGRS86]. Flat Concurrent Prolog ("FCP") is a dialect of CP which supports AND-parallelism, but not OR-parallelism. Also, FCP guards can only contain system-defined test predicates. However, many CP programs are also valid FCP programs. A meta-interpreter for CP can be expressed in FCP. As well, CP programs can be transformed to equivalent FCP programs [Bloc84]. The user environment available for FCP, Logix [SHHS86], facilitates the development and execution of programs. Prototype FCP simulators also exist for multi-processor architectures [TaSS87]. Working implementations in this thesis are therefore developed in FCP.

The discussions to follow assume a familiarity with CP. An in-depth description, including a computational model, is provided by Shapiro [Shap83a, Shap87]. Papers by Shapiro [Shap83c, Shap86b] and Shapiro and Takeuchi [ShTa83] provide summaries. A condensed introduction is also given in Appendix A. (Readers not familiar with CP are strongly urged to read it before proceeding.)

The original semantics of CP [Shap83a] were ambiguous with respect to unification of read-only variables. This led to many problems [Sara85, Ueda85b]. However, a more recent definition of the semantics [Shap86b] rectifies them. Nevertheless, situations which manifested the problems are avoided in this work. As well, no peculiarities of CP are utilized. The ideas in this paper are consistent with, and

applicable to, other parallel logic-programming languages, such as PARLOG or GHC [Ueda85a, Ueda86a]. For example, many of the described programs were also implemented in PARLOG [FoKu86].

CP follows "Edinburgh Prolog" syntax conventions [ClMe81]: constants and functor names begin with a lowercase letter, and variables begin with an underscore character, `_', or a capital letter. The notation *functor/2* indicates a functional term with functor *functor* and arity 2. (Arity *n* means that the function has *n* arguments.) In program examples, all characters from a `%' to the end-of-line are taken as comments.

Logic programs can be interpreted procedurally [Kowa79]. In CP (and FCP), a procedure (definition) is a contiguous list of clauses whose heads have the same name and arity.

As originally defined, CP does not include a sequential-AND construct; only dataflow facilities are available for process synchronization. However, for this work it is assumed that CP contains sequential-AND, denoted '*&*'. Its use is restricted to cases where other sequencing constructs [Kusa84b] would obscure the given example. As in PARLOG [ClGr86], we assume parallel-AND, ',', binds more tightly than '*&*'. Hence

a , b & c , d

is interpreted as

(a, b) & (c, d)

It is straightforward to add sequential-AND using a meta-interpreter and partial evaluator [CoSh86].

Several language enchancements provided by Logix are utilized in subsequent chapters. These are "remote procedure calls" and "modules" [SHHS86]. They are explained in Appendix B.

To avoid time dependencies in specification and implementation programs meta-logical predicates such as *var* and *nonvar* are avoided [StSh86]. Read-only annotations and the *wait/1* predicate, which yield dataflow synchronization, are preferentially used instead.

2.2. Hardware Architecture and Machine Characteristics

The basis for the logic computer system is a CP machine. Such hardware does not exist. Hence it is necessary to define and assume hardware properties and capabilities. This section describes the assumed architecture and general hardware characteristics. Logic inference machines are still in the

very early stages of their evolution. Many architectural proposals exist in the literature with little consensus on their relative merits. Hence, the hardware description is abstract, de-emphasizing specific technical detail. The assumed capabilities are not, however, unreasonable in light of existing, prototypic, or proposed hardware.

As demonstrated by the ICOT's PSI [YYTN83], a logic inference machine can display conventional features, such as interrupts, sequential execution, and reliance on side-effects. Yet it is only natural for a logic inference machine to have higher-level capabilities. Logic-programming languages are high-level languages. A machine which executes such a language should have greater capabilities and complexity. This again is demonstrated by PSI which has firmware instructions to handle process switching, creation, deletion, and synchronization [UYYT83].

The assumed general hardware characteristics are as follows. A CP machine supports the basic computational steps of the language: (read-only) unification, clause selection, and goal reduction. The computer has a multi-processor configuration, consisting of an arbitrary number of individual processing elements. An individual processing element ("PE") is capable of performing complete goal reduction cycles. Each process of a conjunctive goal system can be thought of as executing on an individual processor. "Generic processors" can undertake the reduction of arbitrary goals. The mapping of processes (goals) to processors can be automatic, or user-specified by special notations[2] [Shap83b, Shap84b]. Hardware supports the efficient access and propagation of shared variable bindings. In this discussion a "processor" is a generic processor, unless stated otherwise.

Each physical peripheral device has associated with it a special **device processor.** This processor provides the interface between the remainder of the inference machine and the device. The execution of the device processor is such that all interactions with the peripheral device are describable by a logic program without recourse to side-effects. Viewed by other PEs, the device processor is seen to execute this program. The logic program describes a perpetual process, typically with the state or content of the peripheral device represented as a local argument. This process is called the **device process** ("DP") for the peripheral device. It is indistinguishable from other CP processes. The DP models the transfer of

2. Turtle notations map processes to virtual processors. These virtual processors are supported by physical processing elements through meta-interpretation [TaAS87]. Partial evaluation removes many inefficiencies in the mechanism.

information to and from the peripheral device, usually through unification. A particular processing element may support one or more device processes, or, in general, a mixture of device processes and "normal" processes[3]. However, the mapping of device processes to device processors is constant; DPs do not migrate. A device process exists independently of the operating system processes (see next section); it exists whenever its device processor is active. Software access to a peripheral device is achieved by message passing with the corresponding DP using streams. Device processes may differ in individual protocols, number and type of incident streams, etc.

Conceptually, the nature of the device processor and the software it executes is unimportant. A device processor may be of any type, even von Neumann. It must, however, support an interface consistent with the remainder of the CP machine. Operation of the device processor must be seen to correspond to the execution of a CP program. The nature of that program is dependent on the specification of the entire logic computer system (as discussed in Chapter 3).

The machine language of the CP machine may be CP, in which case its operation is describable by a meta-interpreter. Alternatively, the hardware may execute a logic-based language through which higher-level logic-programming languages (for systems programming) can be implemented (cf. ESP and KL0 [Chik83]). A CP interpreter is then written in this low-level language, but viewed as part of the machine. In either case, unification and goal reduction are provided by a metalevel within the machine model.

A concept similar to device processes appears in Hoare's CSP paradigm [Hoar78]. There the behaviour of special-purpose devices (e.g. I/O devices) is described by CSP processes. Such processes are to be implemented in hardware.

2.3. Operating System Characteristics

An operating system is a collection of software which assists and guides computer hardware in the performance of its tasks, and augments it by providing support functions for users and user programs. An operating system can be regarded as providing an enhanced, abstract machine built upon the underlying hardware. A logic-based operating system fulfills this role for a logic inference machine[4].

3. This is a relaxation of an earlier model [Kusa85b] in which the device processor was dedicated to a single device process.
4. A von Neumann computer can emulate the operation of a logic inference machine by executing

The operating system design assumed for this work follows principles of multi-process structuring (program structuring using multiple concurrent processes). An operating system is composed of small, complementary, and cooperating servers. Each provides a compact set of related services. Servers are constructed as CP objects [ShTa83] or as object hierarchies. They may dynamically create and destroy constituent processes. Servers communicate via object-based protocols using message-passing over streams. They may call on the utilities of devices and other servers in their operation. Servers may be **transient** (dynamically created to fill a temporary need, then removed) or **permanent** (created at system initialization for the duration of system execution). Progressively more substantive services are provided in this manner. Clients normally communicate directly with the server responsible for the utility being sought. Multiple, simultaneous "user tasks" are supported by the operating system.

In a conventional operating system, the bulk of the software cannot access I/O hardware directly. Communication with peripherals involves interrupts. A device driver is introduced to provide an interface. Here, directly accessible devices are provided by the machine model. Clients may access a peripheral device by communicating with its corresponding device process. The operating system need only assist in identifying the appropriate stream. Servers are typically present to provide access control, an alternate interface, additional functionality, etc. During system initialization (discussed in Section 3.2), the operating system obtains a channel to each device process. These channels are preserved for the duration of system execution.

The operating system does not include a kernel. Many capabilities of a conventional "minimal" kernel are already captured by a level of meta-interpretation (see Section 2.2) considered part of the machine model. A process abstraction (process creation, execution, and destruction) is given by goal reduction. Unification provides communication, data transfer, and synchronization mechanisms. Kernel services are obtained without need of "traps" or "(software) interrupts".

For message-oriented operating systems using conventional programming languages and architectures, interprocess communications (IPC) are basic, irreducible (primitive) operations. In CP, however, there is only goal reduction and unification, and IPC is a higher-level, secondary concept achieved by

---

suitable software (an emulator or interpreter). Hence, it is also possible to have a logic-based operating system applied to an "ordinary" computer.

programming technique and convention.

The operating system design makes extensive use of stream-based communication. Some form of stream merging is thus necessary. This may be managed and supported in a manner similar to that employed in LOGIX [HiSS86]. The method entails the automatic enveloping of processes to make merging transparent. A special server, the "stream server", also assists by providing streams to named entities.

This form of logic operating system is not geared toward a particular proposed or prototype inference machine. Rather, it only presumes the previous basic hardware characteristics. Any CP machine, or emulator, having these characteristics should be capable of executing an operating system program following this design.

The Logix user / development environment [SHHS86] can be considered an operating system. As such, it is consistent with many of the design principles outlined here: it makes extensive use of servers, lightweight processes, stream-based communication, etc. Another user / development environment, the PARLOG Programming System (PPS) [Fost87a], follows a somewhat different design [Fost87b]. That design advocates greater control duties for an operating system, and requires language extensions for realization.

Many parallels exist between this logic operating system design and multiprocess-structured operating systems for von Neumann machines, such as Verex [Cher79, Lock79] or V-System[5] [ChZw83, ChZw84]. However, the resemblance is not contrived. The more conventional operating systems were developed using principles such as "light-weight" processes, a minimal kernel, efficient interprocess communication, dynamic process creation and destruction, and groups of processes sharing a common address space. These principles are consistent with the features of CP.

2.4. Concluding Remarks

This chapter has motivated a choice of Concurrent Prolog (CP) as the specification, implementation, and machine language for the thesis. Descriptions have been given of the machine architecture and operating system design for the logic-based computer systems which are the subject of the next chapters.

---

5. Detailed comparisons with Verex and like systems have appeared earlier [Kusa85a].

Merits of these choices will become evident throughout the remainder of the work. The value of many features of CP have already been made apparent.

3. Specification and Initialization

An important aspect of logic programs is that clauses have both declarative and operational readings [Kowa79]. Thus, the same logic-programming language can be used for specification and for implementation; the axioms which describe a model form a program implementing the model. Although this feature of logic programs is often cited in the literature, it is much less frequently exploited.

Typically, specification languages are different from implementation languages. Complex and cumbersome transformations may be needed to realize a working program from a specification. Executable specifications – specifications which can be executed directly as given to form an implementation – offer a preferable alternative.

Formalisms supporting executable specification are most often used for description and verification of hardware components [Jone84b, PaSE85, Suzu85, Turn84, UeKa83]. They are less frequently applied to large, complex software systems, higher-level computer organization, or an overall computer system itself. For example, presentations of ICOT's prototype inference machine PSI [UYYT83, YYTN83] and its operating system, SIMPOS [HaYo83, TYUK84], do not make use of logic programs in their high-level descriptions.

The expressive power of logic programs allows many conventional notions and constructs to be adapted for development of logic-inference machines and their operating systems. Principles of object-oriented programming are one example [Kahn82, ShTa83, Zani84]. However, the unique properties of logic-programming languages suggest that more novel techniques and ideals are also possible. These deserve to be identified and exploited. Further, not all traditional concepts are accommodated well within a "logical" context. Examples include interrupts, synchronization by destructive assignment to "locks", and operations performed by side-effect. In such cases, alternate approaches facilitated by logic programs are preferable. In this chapter, such novel ideas are sought, applied first to the highest levels of a computer system.

This chapter presents models for a logic-based computer system and its two constituents, a logic operating system and a logic-inference machine. The language Concurrent Prolog (CP) serves as the single implementation, specification, and machine language. The computer system is represented as a logic-programming goal

computer_system .

Specification of the system corresponds to resolution of this goal. Clauses used to solve the goal – and ensuing subgoals – progressively refine the machine, operating system, and computer system models. In addition, the accumulation of all clauses describing the logic operating system constitute its implementation. Logic computer systems with vastly different fundamental characteristics can be concisely specified in this manner. Two contrasting examples are given and discussed. An important characteristic of both peripheral devices and the overall computer system – whether they are restartable or perpetual – is examined. As well, a method for operational initialization of the logic computer system is presented. The same clauses which incrementally specify characteristics of the computer system also describe the manner in which this initialization takes place.

Language characteristics utilized within the work – guards, dataflow control, etc. – are not peculiar to CP. They are present in other concurrent logic-programming languages as well. Consequently, the ideas developed in this paper are applicable to related languages such as PARLOG [ClGr86] and GHC [Ueda85a].

The chapter is organized as follows. The high-level, top-down specification of logic-based computer systems is the subject of Section 3.1. "Perpetual" and "restartable" systems and components are introduced and compared. Techniques for representing errors and re-initialization (after error) are explored. Use of an "oracle" process is investigated. Section 3.2 illustrates how the initialization of a restartable logic-based computer system can be declaratively described. Section 3.3 outlines some previous, related work. The chapter is summarized and concluded with Section 3.4. Possible further work is also discussed.

This material, with the exception of Section 3.1.6, has appeared before [Kusa86].

3.1. Specification of the Logic Computer System

A logic-based computer system model has two components, a hardware (machine) model and an operating system model. The hardware model is an extension of the computational model of the chosen logic-based machine language, in this case CP. The operating system builds upon the hardware model. General characteristics of these models were outlined in Section 2.3. They are developed in more detail here with the aid of a concurrent logic-programming language.

3.1.1. Restartable System

A logic-based computer system can be represented by a goal

computer_system.

Specification of the system can be viewed as resolution of this goal. The proof steps represent progressive refinements in the operating system and hardware models. To illustrate, the following clauses could describe the overall system:

```
computer_system :-
   storage_medium( StorageStrm? ),
   tty_keyboard( TtyKeyStrm? ),
   tty_display( TtyDispStrm? ),
   operating_system( [StorageStrm,TtyKeyStrm,TtyDispStrm] ) |
   true.
computer_system :-
   otherwise | computer_system.
```

Figure 1:
Program (a) – Logic-Based Computer System Specification

This concise program specifies the components of the system, and the existence and style of communication channels amongst them. As discussed later, it also describes operational characteristics of system initialization. Clauses for the subgoals

```
storage_medium(StorageStrm?),
tty_keyboard(TtyKeyStrm?),
tty_display(TtyDispStrm?),
```

provide more detail regarding these components of the hardware model. The operating system model is further developed by clauses for

operating_system(DeviceStrmList)

(see Program (e) of Section 3.2, for example). The accumulation of these latter clauses forms a program which implements the operating system.

Execution of the computer system corresponds to construction of the search tree rooted with goal

computer_system .

By nature of the application, the search (execution) never terminates successfully; a computer system is intended to be always executing.

Program (a) succinctly specifies many properties of a logic computer system:

1) The components of the system are a nonvolatile secondary-storage device (disk), terminal keyboard,

terminal display, and operating system, all functioning simultaneously.

2) Communication between each device and the operating system is over a separate stream. Message transfer is initiated by the operating system. Given the most intuitive producer / consumer assignments in the relationships among the system components, individual exchanges are "eager" [ClGr84a, HiCF84, TaFu83] between operating system and storage medium, and operating system and display. They are "lazy" between operating system and keyboard. For example, the producer, *operating_system*, is not constrained in its production of messages by the consumer, *storage_medium*. In contrast, producer *tty_keyboard* is restricted by consumer *operating_system*: *tty_keyboard* cannot send a communication to *operating_system* until the latter instantiates *TtyKeyStrm* to a term, probably of the form *[Msg/TtyKeyStrm']*. Despite the one-way nature of the communication channels, replies can be realized easily using incomplete messages [Shap83c, ShTa83].

3) Subgoals describing devices and the operating system are placed within the guard of a clause. This guard system represents the computation normally being executed. Failure of one of these subgoals causes the resolution of the entire guard to be abandoned, and computation to proceed using the alternate clause (the semantics of *otherwise*) [ShTa83]. This other clause, however, simply re-invokes the goal

   computer_system

   restarting the previous computation, and hence the entire computer system. The system is thus said to "restart (reboot) on failure". It is **restartable.**

4) Since the *storage_medium*, *tty_keyboard*, *tty_display* and *operating_system* subgoals are within a guard, failure of any of them causes the abandonment of the entire computation represented by the clause. Prior to failure, changes to the state (contents) of devices are reflected in the histories (streams) bound to the subgoals' single arguments. Upon failure, results of the attempt to resolve the guard, including the bindings of these streams, are all abandoned. When the second clause succeeds, the resolution of

   computer_system

   begins again, but as if the computation preceding the error had never taken place. To correctly

reflect the correspondence between procedural and declarative semantics, devices must therefore be "restarted". This means operationally resetting the device to the state it was in at the start of the (eventually failing) guard computation (or an equivalent). Hence, devices are **restartable.**

5) A grave software error which results in failure of the *operating_system* subgoal – an operating system "crash" – causes the system to be re-initialized as described in 3) and 4) [Shap83c].

6) Serious hardware errors, which would be expected to operationally require re-initialization of the system, can cause exactly that: they can be treated as failure of the goal representing the malfunctioning device. The effect on the system is demonstrated in 4) and 5) above. Thus hardware errors can be handled cleanly within the logical framework.

7) Power failure, whether deliberate or unforeseen, can be treated as serious hardware error. Another subgoal,

   power_always_up

   could be added to the first guard system in Program (a). Resolution of this predicate suspends while adequate power levels are sustained, but fails if they decline.

8) A manual restart capability (for control by humans) could be implemented as temporary cessation of power, or as a separate signal taken as indicating goal failure.

3.1.2. Perpetual System

Alternate specifications for a logic-based computer system are possible. Some obvious ones are variations on Program (a). For example, each subgoal representing a device could have the initial state of the device as an extra argument.

A computer system of greater contrast to that in Section 3.1.1 is specified by the following clause for resolution of the *computer_system* goal:

```
computer_system :–
   storage_medium( StorageIn, StorageOut? ),
   tty_keyboard( TtyKeyIn, TtyKeyOut? ),
   tty_display( TtyDispIn, TtyDispOut? ),
   operating_system( [StorageIn?, StorageOut, TtyKeyIn?, TtyKeyOut, TtyDispIn?, TtyDispOut] ).
```

Figure 2:
Program (b) – Alternate Logic Computer System Specification

Though not immediately obvious, this computer system is significantly different. In particular:

9) Communication between operating system and devices is still over CP streams. However, separate input and output streams exist. Further, the placement of read-only annotations (assuming that the variables names reflect the true input and output arguments of the device processes) implies that the generation of messages is "lazy". For instance, a request cannot be sent to *storage_medium* until the device process partially instantiates *StorageIn*. Similarly, *storage_medium* cannot generate an output message until *operating_system* or one of its subprocesses partially instantiates *StorageOut*.

10) There is no provision for re-initialization; failure of a goal means failure and termination of the entire system. Once initiated, the computer system is **perpetual.**

11) Devices are **perpetual;** that is, the result of the computation is never "undone" as in the case of a restartable device (see item 4)). Certain devices, such as file storage, are naturally conceptualized as perpetual. A logic-based computer system which has any perpetual component must itself be perpetual.

12) The subgoal representing the operating system cannot be allowed to fail (see item 10). Therefore, the operating system must be very robust and able to always intercept subgoal failure. Some techniques for this are known (cf. failure within a user shell program [ClGr84b, Shap83c]).

13) As the system is perpetual, hardware errors cannot be treated as high-level goal failure. They can, however, be represented by suspension[6]. For example, if a hardware error occurs in a device, it may be treated as suspension of the goal reduction representing the device. Operationally, it is the responsibility of the offending physical hardware to re-establish its state to that immediately preceding the error before the device computation can be seen to continue.

As demonstrated, different characteristics are possible for the logic-based computer system. These characteristics are concisely specified by CP programs. The two computer system examples are consistent with the hardware and operating system descriptions given in Chapter 2. Programs (a) and (b) can be executed as given to simulate the specified computer systems.

---

6. The suspension of goal reduction is a fundamental capability in CP. In fact, Shapiro's original computational model for CP [Shap83a] treats goal failure as infinite suspension.

3.1.3. Problematic Combinations of Properties

Certain combinations of properties of the previous two programs are problematic. For example, a naive mixture of restartable and perpetual devices is not viable because of the commit operator's effect on the propagation of variable bindings. That is, in Program (c)

```
computer_system :-                                                    (c1)
   restartable_part( CommonStrms ),
   perpetual_part( CommonStrms ).

restartable_part( [StorageStrm] ) :-                                  (c2)
   tty_keyboard( TtyKeyStrm? ),
   tty_display( TtyDispStrm? ),
   operating_system( [StorageStrm,TtyKeyStrm,TtyDispStrm] ) |
   true.
restartable_part( CommonStrms ) :-                                    (c3)
   otherwise | restartable_part( CommonStrms ).

perpetual_part( [StorageStrm] ) :-                                    (c4)
   storage_medium( StorageStrm? ).
```

Figure 3:
Program (c) – Anomalous Logic Computer System Specification

any bindings made to *StorageStrm* by resolution of the goal

operating_system([StorageStrm,TtyKeyStrm,TtyDispStrm])

will not be known to *storage_medium* prior to commitment to a clause for solving

restartable_part([StorageStrm]) .

But commitment is not expected in the case of clause (c2) – it is expected that execution of its guard subgoals will continue indefinitely. Clause (c3) has a guard of *otherwise*. Hence, it can be used to reduce the *restartable_part* goal only after the guard computation in (b2), which generates bindings for *StorageStrm*, has failed. Therefore, the *storage_medium* process never receives any messages. Making

operating_system([StorageStrm,TtyKeyStrm,TtyDispStrm])

a subgoal of

perpetual_part([StorageStrm])

alters the symptoms, but does not rectify the underlying problem.

3.1.4. Representation of Errors

Two schemes for representing hardware errors have been suggested: as suspension or as goal failure. These deserve comparison. Other error representation schemes are possible, as well.

Since the CP machine has a multi-processor architecture, representing a hardware error as suspension has advantages over representing it as goal failure. With the goal-failure scheme, knowledge of an error cannot remain local and must be distributed to, and acted upon by, processors responsible for other goals of the conjunctive system. The error-as-suspension approach allows knowledge of an error occurrence to remain restricted to a single device processor.

Error-as-suspension was introduced in the context of perpetual devices. It can also be used for restartable devices, in particular for less serious errors. For example, the computer system should not be re-initialized just because the hard copy printer is suddenly out of paper. It is preferable to have the device suspended in its response to the message which motivated the error condition. The device process will be seen to continue after the problem is solved; for example, when paper is added.

In the restartable system of Program (a), hardware error could have been represented by success of a special subgoal

hardware_error .

Program (a) would become:

```
computer_system :-
   storage_medium( StorageStrm? ),
   tty_keyboard( TtyKeyStrm? ),
   tty_display( TtyDispStrm? ),
   operating_system( [StorageStrm,TtyKeyStrm,TtyDispStrm] ) |
   true.
computer_system :-
   hardware_error | computer_system.
```

Figure 4:
Program (a') – Error-Representation Extension of Program (a)

A possible definition for *hardware_error* is

```
hardware_error :- hardware_error | true.
hardware_error.
```

Given nondeterministic choice of clauses, the subgoal

hardware_error

succeeds in an indeterminate amount of time. When it does, it causes the system to be restarted. Thus, an error is represented as (goal) success. The concept is counter-intuitive, though promising.

Errors of less gravity can be handled by special replies for both restartable and perpetual devices. For instance, output of character *Char* may be achieved by sending the message *out(Char,Reply)* to the

terminal display. To indicate a problem, the terminal display could bind *Reply* to the constant *error* or *false*. A reply of *ok* or *true* might indicate success.

3.1.5. Perpetual versus Restartable Devices

Some peripheral devices are more naturally conceptualized as perpetual devices. For example, with file-structured secondary storage the most up-to-date state (content) should always be maintained. Restartable secondary storage would require that on re-initialization the entire informational content of the device be eliminated, reverting back to some initial state. However, to be useful file storage must be nonvolatile across hardware error, power failure, and other sources of re-initialization. It seems best, then, that a file storage device be perpetual.

Most other peripheral devices can be conceptualized as either perpetual or restartable. For example, on re-initialization the screen of a restartable terminal display can be cleared, reestablishing an initial state. A restartable line printer can generate a page eject to ensure that any output will be at the top of clean paper. However, even though characters have disappeared from the screen, they were present at some specific time with certain characters preceding and following. Similarly, the fact that the line printer generated a particular page of output cannot be later refuted. Therefore, in a more abstract sense, these devices can also be regarded as perpetual. Further, the initial state of the restartable form can be seen as an equivalence class of states of the perpetual form. For a restartable terminal display, for instance, the initial state could be the equivalence class of states of the perpetual form characterized by a clear screen.

The initial states of restartable devices are not restricted to those given in the examples. Instead of a clear screen, the initial state of a terminal display could, for instance, involve having the string "wake me" displayed in the lower right corner. The initial state of a restartable file storage device could include predetermined files and their contents.

A perpetual computer system can be specified in many concurrent logic-programming languages, such as Flat Concurrent Prolog [MTSL85], IC-Prolog [ClMG82], and PARLOG [ClGr84a]. However, a restartable system cannot be specified if, like FCP, the language does not support concurrent execution of guard goals, or if, as in GHC [Ueda86a], only select types of subgoals can appear in guards.

3.1.6. Oracles

As already mentioned, certain peripheral devices are best conceptualized as perpetual. The restartable form of such a device requires the restoration of the last consistent device state prior to disruption. This difficult requirement can be satisfied through an "oracle".

An **oracle** is a nondeterministic computation which always gives the correct answer to a question taken from a restricted set of possibilities. In this context, an **oracle process** can correctly guess the last consistent state of a device. The predicate *oracle* has a single argument and might be defined as follows:

```
oracle( Term ) :- possible_term( Term ).

possible_term( Term ) :- possible_list( Term ).        % true if Term is a list
possible_term( Term ) :- possible_constant( Term ).% true if Term is a constant
possible_term( Term ) :- possible_variable( Term ).% true if Term is a variable
possible_term( Term ) :- possible_compound_term( Term ).
                                                      % true if Term is a compound or structured term

possible_list( [Term|List] ) :-
   possible_term( Term ),
   possible_list( List ).
possible_list( [] ).
```

Figure 5:
Program (d) – Oracle Process

possible_constant/1 and *possible_compound_term/1* are proper relations; e.g.

```
possible_constant( Var )
```

where *Var* is a variable will succeed with *Var* being bound to an arbitrary constant. Thus, their definitions either take advantage of extra-logical predicates such as *functor/3*, *arg/2*, `=..´ and are finite, or (don't use extra-logical facilities and) are infinite. *possible_variable/1* is metalogical, defined using the common Prolog predicate *var/1*. Because of the nondeterministic clause selection, *oracle/1*, when called with a variable as an argument, will succeed with the argument bound to any term. It is indeterminate which.

When called with a variable argument, the oracle computation may correspond only loosely to Program (d). In particular, it may arrive at a term based on residual electromagnetic or mechanical properties from previous activity of the device. Or, it may return a term dictated by human intervention. Because it is a nondeterministic process, it can do so without contravening the declarative meaning of the predicate *oracle*.

A file-structured, secondary storage medium is a device which is best conceptualized as perpetual. However, a restartable form may take advantage of a nondeterministic oracle process to correctly guess the previous state of the storage device. The clause for solving the *storage_medium/1* subgoal in Program (a) might therefore be

```
storage_medium( [init( storage_medium )|ReqStrm] ) :-
   oracle( StorageSystemState ),
   storage_medium( ReqStrm?, StorageSystemState? ).
```

(The motivation for the *init(_)* message is discussed in the next section.) Thus, a storage medium is modelled as starting with some arbitrary state. It is not inconsistent if this corresponds to the state prior to interrupted previous activity.

3.2. Initialization of a Logic-Based Computer System

A device process exists independently of the operating system; it exists whenever its device processor is active. The purpose of system initialization is to initiate the permanent servers (see Section 2.2) of the operating system, and establish communication channels to each device process. Communication via shared variables is declaratively simple (see Programs (a) and (b)). Establishing the shared variables is more difficult.

The following is a simple but effective mechanism for establishing communication channels from the operating system to each device process. It assumes that the CP machine represents variables as pointers into a memory accessible by all processors and globally addressable (though not necessarily global, multi-ported memory). The logic computer system is taken to be of the style in Program (a); in particular, devices are restartable, a single stream exists to each device, and communications are initiated by the operating system. Finally, it is assumed that each device process is able to accept a message of the form *init(DeviceType)* and respond by unifying *DeviceType* with a ground term identifying its type.

N I/O devices, *device1* through *deviceN*, are assumed to exist. The "number" of each device is preset. On initialization, each device processor has a separate, predetermined variable that it tries to access, waiting for it to be instantiated. The first *N* variable addresses are used by the *N* devices. Device *i* tries to access the *i*th variable. The operation of each device – *deviceM* is used as an example – at this point is describable as resolution of the goal

deviceM(DeviceM?) .

Typically this resolution is suspended, awaiting further instantiation of *DeviceM*; i.e. the device is awaiting input from the operating system.

One processor, not a device processor, is designated the **initialization processor**[7]. It is to resolve the subgoal representing the operating system in the overall computer system specification. A representation of the goal is stored in firmware. As Program (b) is the specification, the initialization processor begins resolution of the goal:

operating_system([Device1, . . . , DeviceM, . . . , DeviceN]).

The variable addresses for *Device1* through *DeviceN* are known by the previous convention. However, the operating system does not presuppose which variable will be used for which device.

Computation proceeds according to Program (e).

7. This designation and the number of devices, *N*, can be set in a variety of ways, from firmware memory values to hardware jumpers.

```
operating_system( DeviceStrmList ) :-
   init_server( DeviceStrmList?, DeviceResp ),
   permanent_servers( DeviceResp? ).

init_server( [DeviceStrm|DeviceStrmList], DeviceResp ) :-
   establish_comm( DeviceStrm, EstCommStrm ),
   merge( EstCommStrm?, RespStrm?, DeviceResp ),
   init_server( DeviceStrmList?, RespStrm ).
init_server( [], [] ).

establish_comm( DeviceStrm, RespStrm ) :-
   send( init( DeviceType ), DeviceStrm, NDeviceStrm ),
   establish_comm( DeviceType?, NDeviceStrm, RespStrm ).
establish_comm( DeviceType, DeviceStrm, RespStrm ) :-
   wait( DeviceType ) |
   send( register_device( DeviceType, DeviceStrm ), RespStrm, [] ).

permanent_servers( RespStrm ) :-
   file_system_servers( FSServerStrm ),
   . . .
   user_servers( UserServerStrm ),
   merge( [ReqStrm?,FSServerStrm?, . . . , UserServerStrm?], StrmServerReq ),
   stream_server( StrmServerReq?, [] ).

stream_server( [register_device( DeviceType, DeviceStrm )|ReqStrm], ServerDB ) :-
   stream_server( ReqStrm?, [stream_to(DeviceType?,DeviceStrm)|ServerDB] ).
.
.
.
```

Figure 6:
Program (e) – Operating System Initialization

The original *operating system* goal reduces to the subgoals

```
init_server( [Device1, . . . , DeviceN], DeviceResp ),
permanent_servers( DeviceResp? ).
```

Resolution of the latter subgoal invokes the permanent servers of the operating system. Solution of the first recursively generates *N* subgoals to establish communications. For each device there is a subgoal of the form

```
establish_comm( DeviceStrm, EstCommStrm ) .
```

Consider the general case, device *M*, for which *DeviceStrm* is bound to *DeviceM*. The goal

```
establish_comm( DeviceM, EstCommStrm )
```

reduces to two subgoals

```
send( init( DeviceType ), DeviceM, NDeviceM ),
establish_comm( DeviceType?, NDeviceM, EstCommStrm ).
```

The *send* subgoal succeeds binding *DeviceM* to *[init(DeviceType)|NDeviceM]*. Resolution of the second subgoal suspends awaiting instantiation of *DeviceType*.

It has been prearranged (above) that *DeviceM* is a variable shared by the *deviceM* and *operating_system* processes. Therefore, this new binding of *DeviceM* is also known by *deviceM*. The device processor for device *M* has been awaiting just such an instantiation; i.e. resolution of the goal

deviceM(DeviceM?)

was suspended. By previous assumption the program describing device *M*'s operation contains a clause similar to

```
deviceM( [init( example_type )|ReqStrm] ) :-
  deviceM( ReqStrm?, initial_content ).
```

(*example_type* would actually be replaced by an atom identifying the type of this device; for example, *terminal_display* or *hard_copy_printer*. Likewise, *initial_content* would be replaced by the appropriate initial state for the device.) Reduction of the goal

deviceM([init(DeviceType)|NDeviceM]?)

therefore succeeds, binding *DeviceType* to *example_type*. The suspended goal

establish_comm(DeviceType?, NDeviceM, EstCommStrm)

can now be successfully resolved, resulting in *EstCommStrm* being bound to *[register_device(example_type,NDeviceM)]*. That is, information necessary for further communications with device *M* is placed in a message bound for *stream_server*.

Communication between device *M* and the operating system is now established. The new variable *NDeviceM* is known to both parties and will be used for the next exchange. The establishment of communication at system initialization, then, becomes primarily a matter of coordination.

The following points can be made regarding initialization and Program (e):

1) The clauses for predicates *permanent_servers* and *stream_server* are given in outline form for purposes of discussion. *send*, *merge*, and *wait* are assumed to be self-explanatory, though descriptions are given in Appendix A.

2) The memory locations for variables *Device1* . . . *DeviceN* have no special properties. It should be possible, given sophisticated tail recursion optimization and garbage collection techniques, to reuse them for other variables.

3) Reduction of the subgoals in Program (e) can migrate to idle processors. The initialization processor

is only required to start the computation.

4) A device process need not retain the capability to handle an *init(DeviceType)* message once communications with the operating system processes have been established.

5) Not only does Program (e) specify how system initialization takes place, it is also a refinement of the operating system model. For example, the program initiates, and the operating system is composed of, a set of permanent servers and a transient server to aid in initialization. Certain predicates, such as *establish_comm*, require no further elaboration. The bulk of the operating system is described by the clauses for *permanent_servers* and its subgoals.

6) The high-level specification of the operating system is independent of the number and types of peripheral devices in the computer system.

7) The stream server is an important permanent server. Its purpose is to maintain associations between identifiers (of objects) and communication variables to those objects. On receipt of a message *register_device(DeviceType,DeviceStrm)*, it adds to its database the information "*DeviceStrm* is the stream to device *DeviceType*".

8) The operating system is initiated in such a way that the unexpected absence of a device processor does not create severe problems. The most significant consequence would be an *establish_comm* process suspended, awaiting a reply from a non-existent DP. The rest of the system can carry on. This also means that the operating system can be started expecting more devices than are actually present. New devices can easily be added at a later point in time without restarting operations.

Unfortunately, Program (e) may be too idealistic and impractical given conventional technologies. It is implicit that the entire operating system program is present within the machine at the start of operation. A more typical situation has the operating system program stored on a secondary storage device[8]. The initially executed program – the (primary) bootstrap – is minimal and stored in ROM. Its sole purpose is to read into main memory a larger program and begin its execution. These operational considerations require changes to Program (e) which interfere with the correspondence between model refinement and initialization procedure. However, through a conscious effort and a language extension,

8. Though this is the norm, it need not be. The development of novel computer architectures allows the questioning of such forms of conventional "wisdom".

the interference can be minimized. Program (f) is an example.

```
operating_system( DeviceStrmList ) :-
   boot_server( DeviceStrmList?, DeviceResp, OSProg ),
   prove( permanent_servers( DeviceResp? ), OSProg? ).

boot_server( DeviceStrmList, DeviceResp, OSProg ) :-
   contact_devices( DeviceStrmList, RespStrm ),
   boot_from_storage( RespStrm?, 0, OSProg, DeviceResp ).

contact_devices( [DeviceStrm|DeviceStrmList], DeviceResp ) :-
   establish_comm( DeviceStrm, EstCommStrm ),
   merge( EstCommStrm?, RespStrm?, DeviceResp ),
   contact_devices( DeviceStrmList?, RespStrm ).
contact_devices( [], [] ).

boot_from_storage( [register_device( storage_medium, StorageStrm )|RespStrm], UcrOfOSProg,
                   OSProg, DeviceResp ) :-
   send( query( UcrOfOSProg, OSProg ), StorageStrm, NStorageStrm ),
   send( register_device( storage_medium, NFSDStrm ), DeviceResp, RespStrm ).
boot_from_storage( [Resp|RespStrm], UcrOfOSProg, OSProg, [Resp|DeviceResp] ) :-
   otherwise |
   boot_from_storage( RespStrm?, UcrOfOSProg, OSProg, DeviceResp ).
```

Figure 7:
Program (f) – Operating System Bootstrap

The *contact_devices* process is equivalent to *init_server* of Program (e). The clauses for *establish_comm* are as in Program (e). *otherwise*, *send* and *merge* are familiar CP predicates (a description is given in Appendix A). The new metalogical predicate *prove* is similar to *call* of PARLOG [ClGr84b]. Its definition is an application of the work of Bowen and Kowalski [BoKo82]. Declaratively the goal

prove(Goal, Prog)

succeeds if *Goal* is provable from program *Prog*. Its resolution suspends until both its input arguments are instantiated.

The logic computer system is assumed to include a storage medium device process (to be described in Chapters 4 and 5) in which the remainder of the operating system program is stored. The unique, compact representation (i.e. "location") of that program is preset and known by convention. In Program (f) it is assumed to be *0*.

The role of *boot_from_storage* is to monitor responses from *establish_comm* processes on stream *RespStrm*, watchful for the one identifying the secondary storage device, *storage_medium*. Upon arrival of this response, a query ("read") for the term at the predetermined "location" is sent to the device

process, a replacement *register_device* response is inserted into the output response stream, and the *boot_from_storage* process terminates. All other responses on *RespStrm* are passed unaltered. It is assumed that the term supplied by the storage device contains all clauses necessary for the reduction of the goal

permanent_servers(DeviceResp) ,

i.e. more of the operating system program.

3.3. Other Work

The use of declarative languages in the specification and verification of computers systems, and their components, is often advocated [Turn84]. For example, functional programming languages can be used [Jone84a, PaSE85]. Prolog has also been employed in this capacity [UeKa83]. Several authors [Suzu86, WeSh86] use CP as a specification and verification tool. Their work, however, does not consider software or high-level system characteristics. Other applications of logic programs for specification and implementation include graphics displays [Davi82, HeMa86].

Shapiro [Shap83c] has implemented a number of common, representative operating system functions in CP. He gives, by way of a concise CP program, a high-level specification of an operating system with a "reboot" capability. However, implications for hardware characteristics are not examined. In the same paper, Shapiro proposes that communication with peripheral devices in a CP machine is best achieved by having devices consume or generate CP streams.

In this chapter, the highest level of the operating system specification also corresponds to the steps taken to initialize the system. This is in contrast to the initialization sequence for the FCP machine emulator and Logix user environment [ShSM86]. In the latter, initialization is specified in FCP. It is more efficient, but does not specify as well the components or nature of the operating system and computer system.

The bootstrap technique given here resembles that developed for creating an initial virtual machine on a parallel FCP emulator [TaAS87, TaSS87]. Both involve coordinated access by processors to predetermined shared variables. However, this other initialization procedure does not involve peripheral devices.

3.4. Concluding Remarks

This chapter has presented models for a logic computer system, and its hardware and software components. It has demonstrated that CP programs can be used to concisely specify a logic computer system, its operating system, and operation of peripheral devices. Examples with significantly different characteristics were given and compared. More detail regarding the models is presented in the following chapters as clauses to solve goals such as

storage_medium(StorageStrm)

are given. Techniques for representing hardware errors and for re-establishing device states were also addressed.

A method for operationally initializing a logic computer system was presented. As demonstrated, the correspondence between model refinement and operating system initiation need not be adversely affected by the necessity of a bootstrap. In Program (f), most of the complications are contained within the specification of *boot_server*.

It is noteworthy that concepts such as hardware error and re-initialization do not complicate the declarative reading of Programs (a) and (b). These concepts are inherently operational and are handled within that component of the language and computer system models.

3.4.1. Further Study

Several specific topics for further study are immediately apparent:

- Errors can be represented by suspension for both perpetual and restartable devices. However, this requires that, following the error, a device continue from the precise point of preemption. Implementationally, this should not be difficult to approximate. It is not clear, however, how it might be precisely attained.
- Certain devices, such as file storage, are best conceptualized as perpetual. However, hardware errors cannot be represented as conjunctive goal failure for perpetual devices. The error-as-suspension scheme may also be unworkable because of the problem mentioned above. Therefore, other means of handling hardware errors should be investigated.
- Because of Program (c), it may be taken that restartable and perpetual devices cannot both be present within a single logic-based computer system. This conclusion is premature, however. An extended

form of the metalogical predicate *prove* (similar to the "three-argument meta-call" of PARLOG [ClGr84b]) offers several possibilities for combination of both types of system components.

- It may be possible to capture the character of restartable devices through the event calculus [KoSe86]. An event relates a device state to a time (of the event). A "restart" is thus an event which relates a particular time and a state which is known to have existed at some earlier time.

Preliminary investigations suggest interesting results in these areas.

As Programs (a) and (b) suggest, computer systems with a wide variety of characteristics can be specified. As further study, systems with varying properties can be developed, explored, and compared. Techniques for operationally initializing these systems can also be investigated. Other issues which can be explored include protection and security and user-programmable error handling.

4. Devices and Declarative I/O

In the previous chapter, a logic-based computer system, including its component peripheral devices, was described at a high level using a logic-programming language. This chapter pursues the specification of these peripheral components. The goal is to model devices supporting **declarative I/O**, devices whose actions and interactions can be described declaratively.

It is unfortunate to find I/O operations within logic-programming environments achieved by side-effects. For example, resolution of the Prolog goal

put(Char)

places, as a side-effect, the character *Char* on an output medium (usually the user's terminal) [ClMe81]. This perception of I/O has pervaded the development of logic-programmed systems. System predicates having the side-effect of reading(writing) characters from(to) physical I/O devices are common. Very often, all I/O is based on such predicates. This contradicts one of the supposed advantages of logic programming, that it is side-effect free. As well, it is often necessary to convert the information (logical terms) to a sequence of characters before output. Input typically also involves conversion, plus tokenizing and parsing operations. As exemplified by Prolog's `=..´ operator, changes of representation tend to be inefficient. Finally, order of goal reduction is usually critical to correct operation of such predicates. This chapter concerns development of declarative interaction with peripheral devices (declarative I/O) where such conversions are unnecessary and no reliance is made on side-effects.

Instead of considering many peripheral devices with differing characteristics, a representative is chosen. A good choice is the peripheral device providing nonvolatile, high-capacity information storage: a magnetic disk. This is a common, though conceptually complex, component of most computer systems. The device is assumed to be restartable, as discussed in Sections 3.1.1 and 3.1.6. As is shown, the concepts developed for this device are applicable to other devices. In the discussion, a "disk" is any high-speed, nonvolatile, large-capacity, long-term storage component of a computer system.

The operation of a computer system is often critically tied to the methods used to store, retrieve, and manage information. A secondary storage system is therefore of great significance in characterizing a computer system: it is generally indicative of the overall design philosophy; it usually serves as a

basis for other (sub)systems; and in describing it, other aspects of the computer and its design are revealed. The power of logic programs allows the adaptation and translation of existing computational techniques to inference machines. This can also be done for secondary storage. However, the development here is an attempt to better utilize the novel properties of a concurrent logic-programming environment.

Two general techniques for developing a secondary storage device model are presented in this chapter. The first views a computer system as a group of independent logic systems or nodes. A secondary storage device is seen as one such node, its content being a knowledge base. This is the subject of Section 4.1. In the second technique, the disk is viewed as the computation of a function. This is described in Section 4.2. In either case, both infinite and finite storage capacities can be accommodated, and unification describes information retrieval and storage in a natural way. For each general technique, various options are investigated. The similarities and differences between the resulting models are examined in Section 4.3.

According to the assumed hardware architecture (see Section 2.2), operation of a peripheral device is described by a device process (DP). A DP reflects the computation preformed by its device processor. In this chapter, various secondary storage device processes are presented. They are defined in CP. Each captures a separate disk model. A device process (typically) consumes a stream of requests generated by other processes in the logic system. Device state or contents are represented by a local argument, the process's data state. Queries and assertions are processed using this local data. An implementation of the DP must map these logical computations into machine operations involving the actual storage memory. For each model, the device process is named *storage_medium.*

The disk models are specified in CP. Being executable, the specifications can be used as given to provide an inefficient and volatile, but correct, storage medium for a parallel logic-based system. An actual implementation of one model is presented in a subsequent chapter.

4.1. Independent Logic System Approach

A logic-programming language models a computer as a logic database and an inference engine. A declarative interface to peripheral devices can demonstrate like properties.

The **query-the-user** facility [Serg83] is a good starting point. Query-the-user views an interactive inference system as a single inference engine (the computer) and a global database (InternalDB + User). The latter is comprised of both the logic program stored in the computer (InternalDB), and facts known to the user or other external agent (User). The '+' indicates knowledge-base union in the style of O'Keefe's module calculus [OKee85]. The auxiliary database (User) is queried by the logic system when its internal information proves insufficient to answer a query. Meta-logical information specifies which predicates are defined in it. Conceptually, answers to queries are logical deductions from the combined, global database.

Following the query-the-user concept, a magnetic disk might be modelled as an external database (Disk). Queries would then be solved relative to the combined database (InternalDB + Disk). Unfortunately, this is only adequate for retrieval of information where the disk serves as an invariant extension of the internal database. An important property of disk storage is its ability to record changes to information over time. The location of information is thus significant, as is the transfer of information from the external agent to the internal database (or vice-versa). Query-the-user does not model this transfer. The total information content of the global system is always constant. Therefore, a straightforward application of the query-the-user concept is not acceptable. A more general model is necessary.

Such a model is as follows. A logic-based computer system is composed of two or more distinct computational nodes, where each is a logic-based system in its own right. Every node possesses a logic database, inference mechanism, communication mechanism, and specific goals that it seeks to achieve. Because of the hardware paradigm presented in Section 2.2, each node is a Horn-clause inference engine. The inference mechanism can reference only the node's own database. No assumptions are made as to the consistency of the "world views" held by the component logic systems. They are treated as separate entities and significance is given to the passage of information to and from them. A communication represents either (1) a statement by one logic system of what it believes to be true (an **assertion**), or (2) a request for confirmation or denial of a statement (a **query**). Recipient logic systems are free to either reject assertions, or, if they can, assimilate the knowledge they represent into their databases. Such issues as why and how nodes choose to make assertions are not considered in this paper (it is presumably governed by some sort of suitably expressed metalogic). Neither is the physical nature of communication between nodes considered.

The model is simple and general. It maintains an overall logical picture, yet simultaneously gives meaning to data movement. This section investigates a secondary storage medium based on it. The nature of interactions of the storage medium with other components is considered. Two alternative methods for actually storing information (knowledge) are examined. Executable specifications for the two formulations are given in CP. Extensions to the model which allow "garbage collection" of old knowledge are then pursued. An earlier formulation of these ideas has appeared [FoKu86].

4.1.1. General Model

The disk interface must enable other nodes to interact with the disk as if it

- possesses a knowledge base;
- has an inference engine capable of processing assertions and queries made by other nodes; and
- can apply some metalogical constraints on received assertions in order to either assimilate or reject them.

These characteristics can be achieved as follows.

A disk is easily perceived as a knowledge base. Trivially, it contains information. This information can be structured similarly to an "in-memory" knowledge base; that is, as a set of clauses. It is also possible to specify a disk which accepts and transfers clauses. It is more difficult, however, to provide an inference mechanism capable of performing full meta-level knowledge-base maintenance [BoKo82] (checking for redundancy and inconsistency on assertion, removing derived knowledge on retraction, etc.). In principle, such a mechanism can be captured by a highly intelligent disk controller. This requires more computing power than may be considered appropriate for a peripheral device interface. Therefore a much simpler inference mechanism is proposed for the disk — one capable of processing assertions and queries that refer only to facts (unit clauses) defining a single predicate, *disk/1*. The mechanism is not required to deal with retractions. Also, there is no provision for backtracking – in the model or the language – so assertions are nonretractable.

A query of the disk might have the form *query(disk(Term))* where *Term* is an arbitrary term. This query succeeds or fails, depending on whether the disk contains the clause *disk(Term)*. This could be prohibitively inefficient. Evaluation of the query requires, in the absence of any indexing, a serial search over the disk. An indexing mechanism is therefore introduced. One possibility is to use a hashing

function that, given a term, returns a (not necessarily unique) index. This minimizes the searching required to determine whether or not the disk contains a particular *disk(Term)* fact. Such an approach has been used effectly to provide rapid access to PROLOG databases [RaSh86]. However, even the modest overhead of hashing is unnecessary; it is only necessary to generate a simple, unique index. Furthermore, the benefit of hashing is reduced for queries in which *Term* is partially instantiated. Therefore, a hash function is not employed. A table mapping indexes to disk addresses is maintained instead (at least implicitly). As facts are assimilated, unique identifiers are allocated and returned to the originating node. This is described in more detail below. The only restriction on identifiers is that they be ground terms. Without loss of generality, non-negative integers are used in this work.

The assertion of a fact *disk(Term)* leads to the addition of the information *Term* to the knowledge base. A **unique identifier** ("ID") is associated with the new information and returned. The ID is an index into the mapping table and servers as an identifier that other nodes in the logic system may use to refer to the clause in subsequent queries. The actual clause stored is thus *disk(Id,Term)* and the form of an assertion is *assert(disk(Id,Term))*.

Use of this identifier is illustrated by the following example. Suppose that some node communicates the assertion *assert(disk(Id,f(X)))* to the disk node. *Id* is not instantiated. Assuming that the disk decides to accept this assertion, some unique identifier (say *5*) is associated with the term and the clause *disk(5,f(X))* is added to the disk's knowledge base. The identifier (index) *5* is returned to the originating node. This node (or any other to whom the identifier is subsequently communicated) is now able to query the disk as to whether it includes the fact *disk(5,f(X))*; the query *query(disk(5,f(X)))* will succeed.

Other forms of query are possible. For example, *query(disk(5,g(X)))* is legitimate, but fails (the disk does not include the fact *disk(5,g(X))*). To find out what term is associated with a identifier, say *5*, the query *query(disk(5,Term))* is used. It succeeds, revealing that the stored term is *f(X)*. *query(disk(5,f(a)))* also succeeds. However, the variable *X* stored by the disk must not be bound to *a*. The effect, otherwise, would be the query causing to be stored the very information whose validity was being ascertained.

To avoid costly searches, asserted information is not checked for duplication. Thus, a repeated assertion *assert(disk(Id,f(A)))* results in a new clause being added to the disk database, rather than a

search for the previously stored clause *disk(5,f(X))*. A new identifier, say *6*, is returned.

The assimilation of information by the storage medium is governed by a simple consistency constraint: only one fact is associated with each ID. This means that an ID can be considered a **unique, compact representation** ("UCR") of the term with which it is associated. An ID is a unique name that can be used by other nodes of the logic system in place of the associated term. *disk(Id,Term)* expresses that *Id* and *Term* represent the same information. A UCR is useful because it is typically much simpler (and "smaller") than the associated term.

The storage medium content can be represented as clauses defining the predicate *disk*, e.g.

```
disk( 1, first_term ).
disk( 2, second_term( has_structure ) ).
disk( 3, [third,term,is,a,list] ).
```

However, it is more convenient to define *disk* from a meta-level. The definition is given by terms in a single structure kept as a local argument of a perpetual process, say *storage_device*. That is, the same storage configuration above is captured by

```
storage_device( [disk( 1, first_term ),
                 disk( 2, second_term( has_structure ) ),
                 disk( 3, [third,term,is,a,list] )] ).
```

Since each fact has functor *disk*, the functor can be omitted for brevity, and only the arguments stored. Thus, the stored pair (tuple) *(5,f(X))* represents the clause *disk(5,f(X))*. The previous storage specification becomes

```
storage_device( [( 1, first_term ),
                 ( 2, second_term( has_structure ) ),
                 ( 3, [third,term,is,a,list] )] ).
```

Requests *assert(disk(Id,f(X)))* and *query(disk(5,f(X)))* are likewise shortened to *assert(Id,f(X))* and *query(5,f(X))*, respectively.

These ideas can be incorporated into the secondary storage device process, *storage_medium*. The process's data state, which represents the disk content, is the list of *(Id,Term)* pairs. The DP consumes a stream of messages generated by other processes in the computer system. These are the assertions and queries. The messages are processed using the facts in the list of pairs. The DP behaves as a (restricted) knowledge-based inference system.

4.1.2. Shared Variables

assert(disk(Id,f(X))) from the example above contains the free variable *X*. If the term *disk(5,f(X))* is actually stored, the variable *X* becomes **shared** between client and storage nodes. It is not difficult to define a system in which variable bindings are maintained between a disk and elsewhere in a logic system. However, it would be very difficult to efficiently implement. Once communicated, *X* could be instantiated to any arbitrary term by either the originating or receiving node. Propagation of the new binding might involve complex message passing. Some mechanism would also be necessary for noting and incorporating remote instantiations.

Variables shared between nodes compromise the independent nature of the disk. If the disk and client are linked by bindings, it is hard to identify precisely, or assign significance to, the transfer of information from one to the other. Also, it suggests that the two media represent, at least in part, the same world view.

Restrictions must therefore be imposed on the information actually stored on the medium. Two possibilities are considered here:

1. only allow ground terms to be stored on disk; or
2. allow terms containing variables to be transferred, but require that the variables be renamed.

These restrictions ensure that bindings from disk to elsewhere in the system cannot be created. These two approaches differ in subtle, but interesting, ways. They are each considered in turn.

4.1.3. Storage of Ground Terms

To prevent creation of variables shared between disk and client, all information can be "frozen" (made ground) before being stored ("transferred" to the disk content). This option is investigated in this section. The resultant model resembles Kowalski's **logic-based open system** scheme[9] [Kowa85] in style.

A "freeze" operation [NaTU84] converts a (possibly non-ground) term to a ground term in which all variables are replaced with some conventional representation of variables as constants. For example, *f(X,g(Y,X))* might be converted to *f($(1),g($(2),$(1)))*. Freezing corresponds to the action of *number_var*

9. In Kowalski's scheme, nodes can be arbitrary inference machines.

in Edinburgh-style Prolog systems [ClMe81, Quin87]. The reciprocal operation is "melting"; the representations of variables are replaced by new logical variables. The new variables will be different (renamed) from those in the original term. Duplication of variables within an object term is maintained by both freezing and melting.

The storage medium device process incorporates an interface to the remainder of the system. Variables persist into the interface, but are made ground before being incorporated into the data state.

4.1.3.1. Specification of the Device Process

The *storage_medium* process has two arguments. The first is an input stream of requests. The second is the current contents of the disk. Assertions and queries accepted by *storage_medium* have the form *assert(Id,Term)* and *query(Id,Term)*. The device content, represented by *DeviceContent*, is initially *[]*.

```
storage_medium( [assert( Id, Term )|Input], DeviceContent ) :-                (g1)
  freeze( Term, FrozenTerm ),
  allocate_unique_id( FrozenTerm?, Id )|
  storage_medium( Input?, [(Id?,FrozenTerm?)|DeviceContent] ).
                                        % add to content on recursive call
storage_medium( [query( Id, Term )|Input], DeviceContent ) :-                 (g2)
  lookup( Id?, FrozenTerm, DeviceContent? ),      % look for entry
                                        % first input argument must be supplied
  melt( FrozenTerm?, Term )|
  storage_medium( Input?, DeviceContent ).
storage_medium( [IllegalRequest|Input], DeviceContent ) :-                    (g3)
  otherwise|                            % illegal or failing request
  storage_medium( Input?, DeviceContent ).

lookup( Id, Term, [(Id,Term)|DeviceContent] ).    % match found             (g4)
lookup( Id, Term, [NoMatch|DeviceContent] ) :-                                (g5)
  otherwise|                            % no match, keep looking
  lookup( Id, Term, DeviceContent? ).
```

Figure 8:
Program (g) – Storage Medium Specification (ground terms stored)

The first clause for *storage_medium* handles assertions. A new, unique ID is allocated and the supplied term is "frozen". The information is added to the storage medium content by prepending the pair *(Id?,FrozenTerm?)* to the list *DeviceContent*. The read-only annotations prevent instantiations of *Id* or *FrozenTerm* by subsequent queries. Clause (g2) handles queries. A message *query(Id,Term)* succeeds if it is possible to locate a pair *(Id,StoredTerm)*, "melt" the stored term, and unify the melted term with the query term. *melt* is discussed below. Unification may lead to the term component of the query

becoming further instantiated, but it cannot modify the stored information. Binding *Id* in *assert* requests, and *Term* in *query* requests, transmits information to the client nodes. The calls of *freeze*, *melt*, *lookup*, and *allocate_unique_id* are in guards. Hence, clause (g3) allows the process to continue despite failure of a request.

The semantics of *freeze* and *melt* are introduced above. They are user-definable, but may be assumed provided by the language system – either as library predicates or primitive, "built-in" predicates. *freeze(Term,FrozenTerm)* binds *FrozenTerm* with a frozen representation of *Term*. *melt(FrozenTerm,MeltedTerm)* waits until its first argument is instantiated, melts the term, and unifies the result with *MeltedTerm*.

allocate_unique_id must generate a nonce identifier for any given value of its input arguments. One possible definition is analogous to *determine_unique_id* outlined in Section 4.2.1; i.e. a simple "counter". It requires adding a third argument, the last unique ID allocated, to the *storage_medium* predicate. The newly generated ID is the integer one greater than the last-allocated ID. (Recall that for discussion purposes, IDs are restricted to non-negative integers). Other algorithms are certainly possible.

The predicate *lookup(Id,Term,Contents)* attempts to find a pair *(StoredId,StoredTerm)* in the list *DeviceContent* such that *(StoredId,StoredTerm)* and *(Id,Term)* unify. As stated earlier, this retrieval is based on index *Id*. Thus, the implementation of the predicate is not as given by clauses (g4) and (g5). Rather, *lookup/3* succeeds or fails for all sets of bindings to *Id*, *Term*, and *DeviceContent* for which the predicate defined by (g4) and (g5) would succeed or fail (and the same bindings are generated).

The read-only annotation of *Id* in the *lookup* subgoal of clause (g2) requires that the client supply an ID with a query[10]. Otherwise, satisfaction of the query will suspend due to suspending unification of *Id?* with the stored ID in clause (g4). Also, *lookup/3* is invoked with its second argument uninstantiated; i.e. the search is based on the ID. This prevents pattern-directed searches of the disk content based on the *Term* argument of the request.

query and *assert* messages can have varying patterns of instantiated and uninstantiated arguments. An assertion usually has its first argument variable (an output argument). Otherwise, the originating node is "guessing" what ID will be allocated to the fact when it is stored. This is not erroneous, but

10. This applies only if IDs are atomic and not functional terms, as is the case for this discussion.

atypical. For *query(Id,Term)*, *Id* is usually ground (input) and *Term*, variable (output). If *Term* is non-variable, the client is confirming that a certain ID has a particular term (or term structure) associated with it. Again this is unusual, but not problematic. If the usual output argument of either request is given as a read-only variable by the client, unification involving that variable suspends. Subsequently, the storage medium computation suspends. It will not resume until the read-only variable is instantiated by the client. This behavior can be prevented by adding metalogical tests to the guards of clauses (g1) and (g2).

In accordance with the ideas in Section 3.3, the device process is initiated by

```
storage_medium( [init(storage_medium)|Input] ) :-
    oracle( DeviceContent ),                    % nondeterministically choose a
    storage_medium( Input?, DeviceContent? ).   % starting content
```

A specification corresponding to Program (g) can be given in PARLOG [FoKu86]. Because Program (g) makes use of full unification, the PARLOG formulation is longer. In PARLOG a "test unification" predicate is required which can test, in a guard, the unifiability of two terms without generating bindings. This is necessary to ensure "guard safety". To define the predicate, a copy – in which all variables are renamed – must be made of both terms. This copy operation is performed implicitly in CP in supporting multiple OR-parallel environments.

Program (g) specifies an inference system, plus an interface to the remainder of the logic-based system. The head of each *storage_medium* clause and the calls to *freeze* and *melt* are part of this interface. The separation of the process into interface and purely "local" portions becomes more apparent in the next chapter where the specification is transformed in preparation for implementation.

In this secondary storage model, information is frozen before being stored. This eliminates the problem of maintaining shared variables. Additionally, it prevents expansion of already-stored information. Hence, if the disk already holds *disk(5,f(X))*, the stored information cannot be changed to *f(p(Y))*, say, by either the client binding (his copy of) *X* to *p(Y)* or by a message *assert(disk(5,f(p(Y))))*. Resources invested in storing a fact remain constant.

4.1.4. Storage of Copied Terms

Another method for preventing shared variable links between disk and client is renaming variables on assertion and query. This alternative is examined in this section. The resulting storage medium

specification is akin to the previous one. Many of the comments made in Section 4.1.3.1 apply here as well. However, this model allows greater functionality.

A storage medium is again viewed as a node in a larger multiprocessor configuration. It includes an inference mechanism and own knowledge base. As before, the inference mechanism accepts assertions of facts which define the relation *disk*, and queries of these facts. However, variables are renamed before any unifications involving the data state of the process (the disk content). This allows a term containing variables to be stored on disk. Subsequent unification may further instantiate the stored term, but cannot create variables shared between disk and client.

The requests supported by Program (g) are also supported in this scheme. However, the semantics of *assert(Id,Term)* are extended, in particular when *Id* is bound. In that case, the variables of *Term* are renamed. If a fact *disk(Id,StoredTerm)* already exists on the disk such that *Term* and *StoredTerm* unify, the unification (and request) succeed. This may alter the stored information, *StoredTerm*. Otherwise, the request is handled as in Program (g). There are now two types of *assert* request: assertion of new knowledge or refinement of existing knowledge.

The specification of the device process is as follows.

```
storage_medium( [assert( Id, Term )|Input], DeviceContent ) :-                (h1)
   copy( Term, TermCopy ) &
   allocate_unique_id( TermCopy, Id ) |
   storage_medium( Input?, [(Id?,TermCopy?)|DeviceContent], Id? ).
storage_medium( [assert( Id, Term )|Input], Content ) :-                      (h2)
                                          % handle other form of assertion
   copy( Term, TermCopy ) &
   lookup( Id?, TermCopy, Content? ) |
   storage_medium( Input?, Content ).
storage_medium( [query( Id, Term )|Input], DeviceContent ) :-                 (h3)
   lookup( Id?, StoredTerm, DeviceContent? ) &
   copy( StoredTerm, Term ) |
   storage_medium( Input?, DeviceContent, LastId ).
storage_medium( [UnknownRequest|Input], DeviceContent ) :-                    (h4)
   otherwise |
   storage_medium( Input?, DeviceContent, LastId ).

lookup( Id, Term, [(Id,Term)|DeviceContent] ).                                (h5)
lookup( Id, Term, [NoMatch|DeviceContent] ) :-                                (h6)
   otherwise |
   lookup( Id, Term, DeviceContent? ).
```

Figure 9:
Program (h) – Storage Medium Specification (copied terms stored)

Clauses (h1) and (h3) through (h6) correspond closely to Program (g). With the exception of handling renamed terms rather than frozen ones, the meanings of the clauses are the same. *lookup* is exactly as before. The same definition of *allocate_unique_id* can be used. Clause (h2) is the only addition. If it is possible to find stored information for a given ID, the stored term is unified with a copy of the term, *Term*, given in the assertion. No bindings to *Term* are returned to the client as unification is with a copy of *Term*.

Sequential-AND appears in clauses (h1) through (h3). It is necessary to synchronize the guard goals. Read-only annotations are insufficient as variables are legitimate input to the second goal in each case. In Program (g), where stored information is ground (frozen), read-only synchronization was adequate.

copy creates a duplicate – in which all variables are renamed – of its first argument and unifies this with its second argument. If not predefined, it can be provided, among other ways, using *melt* and *freeze*:

```
copy( Term, Copy ) :-
   freeze( Term, Intermediate ) ,
   melt( Intermediate?, Copy ) .
```

By use of auxiliary clauses this specification could be made more efficient; e.g. the copying of *Term* in the guard of each of (h1) and (h2) could be performed but once. However, efficiency gain is not critical and the modified program is more difficult to understand. Hence such modifications for efficiency are not pursued.

The added feature of this model is that stored information can be refined or augmented. Hence, if the disk currently associates the term *f(X)* with ID *5*, the information can be extended to *f(p(Y))*. With the previous model, a new ID would have been required.

4.1.5. Summary and Analysis of the Two Approaches

The disk can be viewed as a separate system with a knowledge base and simple inference mechanism. It is able to process assertions and queries made by other nodes. These must refer to facts defining a single predicate, *disk*. Storage of information on the disk corresponds to assimilating the knowledge in an assertion; retrieval, to answering queries. Unification captures both operations. No provision is made for backtracking; that is, assertions cannot be retracted, once made. An ID serves as a key or index for

rapid retrieval, and as an identifier by which other nodes reference a particular fact in (subsequent) queries. IDs are ground terms. Finally, the knowledge base is considered a closed world; a query with a non-existent ID fails. Two variations in the model have been presented, differing in the form stored information takes.

Programs (g) and (h) operate similarly for queries and assertions in which the ID argument is variable. However, the copying scheme allows assertions in which existing IDs are given as input. Such assertions enhance or extend information already stored without a new ID being allocated.

Both variations prohibit variables shared between disk and client. However, using frozen representations means that the resources necessary to retain a certain fact, once asserted, remain constant. This makes the model specified by Program (g) potentially easier to implement, especially if storage resources are finite. For the second model, additional checks are required to ensure that some local or global capacity is not exceeded.

An interesting aspect of these schemes is the implicit concept of "environment". An environment can be defined as the scope over which knowledge of a variable can be communicated without its being renamed or frozen. Interfaces exist between environments. Information transfer at the interface is by unification. Renaming or freezing of variables also occurs at the interface.

The disk was chosen as a representative case for peripheral devices. Both variations work well for a disk. However, for other devices this may not be the case. A terminal is an example. It would handle assertions solely. Asserting a fact would add it to the terminal's knowledge base, which would correspond to the actual physical display. It is not clear how a variable could be represented on a video display, however. It therefore appears that the first technique, where all asserted information is made ground, is preferable in this case.

The model of Section 4.1.3 predates the others presented. It has been the subject of more work and study [FoKu86]. Hence, the remainder of this section, which discusses extensions to the separate inference engine model, assumes information is frozen when stored by a storage medium device process. This does not imply a statement of strength or preferability between the two approaches. Many of the ideas carry over quite easily to the copying scheme. Such work is left for future investigation.

4.1.6. Monitoring Free Space

The specifications above capture very simple secondary storage systems. In particular, no check is made for exceeding available capacity. Thus they implicitly assume infinite resources. This is acceptable only for extremely high-capacity media such as optical disks.

The *storage_medium* specifications can easily be modified to account for finite capacity. Another argument, indicating the current amount of free space on the disk, is maintained. A predicate is necessary to calculate the "size" of a term to be stored (*freeze* or *copy* can even be extended to do this). Before adding new information to the disk, a check is made for sufficient free space. If the check fails, the assertion can be failed. A test for sufficient storage space can be regarded as a further integrity constraint. These modifications do not permit reuse of space, however. Section 4.1.5 described a further complication regarding finite storage with the copying scheme.

4.1.7. Time-stamps and Other Extensions

Secondary storage systems typically provide more than just nonvolatile storage. They also support the change of information over time. The independent logic system model developed in Section 4.1.3 captures storage of information, but not modification of stored information[11]. Also, the model does not facilitate reclamation of resources. These issues are discussed in this section. File servers are first outlined. Time-stamping is introduced and merged with the previous storage medium concepts to yield a more sophisticated disk model that allows garbage collection of "old" knowledge. This discussion is an abbreviated treatment of that found elsewhere [FoKu86].

4.1.7.1. File Servers

In an earlier presentation of this material [FoKu86], file servers are developed. They utilize a declarative secondary storage analogous to that described in Section 4.1.3. Initially a simplistic file server is specified. In summary, it is a persistent process which maintains the ID corresponding to each (supported) file's current content. It accepts a stream of *read(File,Content)* and *write(File,Content)* requests. A *read* request is translated into a query of the storage medium using the appropriate retained ID. A *write* request motivates an assertion of the new content to the storage medium. On success the

11. Program (h) allows extension of information by further instantiation, but not replacement.

new ID is retained. The entire file content is transferred on each *read* or *write*. (In this work, file servers are addressed in detail in Chapter 6).

The simplistic file server maintains the ID of the most recent version of a file. As with other forms of data, it may sometimes be useful to refer to earlier versions of the same file [ErRa84]. This is possible if the file server is extended so that it maintains a list, or **history**, of time-stamped IDs. The time stamp indicates the time, according to some standard, at which the ID was associated with the file's content. The necessary modifications to the server are discussed in the previous work. Highlights are:

- A *write(File,Content)* request is processed by making an assertion. A time stamp and the new ID are retained in the history, instead of just the ID.
- A *read(File,Content)* request is assumed to refer to the most recent version of a file. The most recent ID (for the file) in the history is selected, and a *query* generated.
- A second form of *read*, *read(File,Content,Time)* is introduced. It requests the value of the file at some previous time. The history is searched for to find the ID current (for the file) at the given time. A query using that ID is then issued.

This formulation retains more information – the history of state changes undergone by a file – and is thus more powerful. The event calculus of Kowalski and Sergot [KoSe86] gives logical meaning to such a sequence of state changes (events).

The enhanced file server suggests a method for garbage collection of stored information. It is difficult to declaratively account for the loss of information (knowledge) once asserted. The recorded time-stamps mean, however, that pragmatic metaknowledge can be applied, such as: "no-one wants to know about states more than a month old". This allows "old" knowledge to be purged from the system, since it is "known" that it will never be referred to again. Purging could involve actual removal of information or just relocation to an archiving medium such as magnetic tape.

4.1.7.2. Extensions to Disk Specification

The inclusion of time-stamps in the file server enhances its expressive power. This suggests that it might be useful to introduce this concept at the storage medium level. In this section the implications of this are considered, and a specification for the resulting extended storage model is given.

The model of the disk is changed. Instead of a knowledge base containing information as facts, it will now contain histories of facts. Its interface to other nodes is also changed: it is able to accept queries about a fact defining *disk* at a particular time. For simplicity, however, it is assumed that other nodes only make assertions concerning the present state. Each *disk/2* fact is considered an entity.

Conceptually, the disk database is now a collection of clauses of the form *disk(Id, Term, Time)*. An assertion *assert(disk(Id, Term))* is interpreted as meaning: "the triple *(Id,Term,CurrentTime)* defines the *disk* predicate as of time *CurrentTime*." *CurrentTime* is a time stamp taken according to some standard for the system. The integrity constraint that there be no more than one *disk* clause for each distinct ID must also be relaxed. There can now be many. However, there is only one for each distinct ID-timestamp combination.

The mechanism used to evaluate a query can be defined metalogically [KoSe86]:

```
query( Id, Term, QueryTime ) IF
  disk( Id, Term, Time ) &
  Time ≤ QueryTime &
  NOT [ disk( Id, SomeOtherTerm, SomeOtherTime ) &
        Time < SomeOtherTime < QueryTime ) ]
```

That is, it can be deduced that the stored term with identifier ID at time TIME is TERM, if we can show that ID was assigned contents TERM at a time TIME' prior to TIME and that no other assignment has occurred between times TIME' and TIME. In practice, efficient application of this mechanism requires that time be incorporated into the indexing of clauses.

An executable specification for a disk that maintains histories of entities follows.

```
storage_medium( [assert(Id,Term)/Input], DeviceContent ) :-                    (i1)
   freeze( Term, FrozenTerm ),
   allocate_unique_id( FrozenTerm?, Id )|
   time( Time ),
   storage_medium( Input?, [(Id?,FrozenTerm?,Time?)/DeviceContent] ) .
storage_medium( [assert(Id,Term)/Input], DeviceContent ) :-                    (i2)
   time( Time ) ,
   lookup( Id?, _, Time?, DeviceContent? )|          % make sure ID exists
   freeze( Term, FrozenTerm ) ,
   storage_medium( Input?, [(Id?,FrozenTerm?,Time?)/DeviceContent], Id? )
storage_medium( [query(Id,Term,Time)/Input], DeviceContent ) :-                (i3)
   lookup( Id?, FrozenTerm, Time?, DeviceContent? ),
   melt( FrozenTerm?, Term )|
   storage_medium( Input?, DeviceContent ).
storage_medium( [query(Id,Term)/Input], DeviceContent ) :-                     (i4)
   time( Time ),
   storage_medium( [query(Id,Term,Time)/Input], DeviceContent ).
storage_medium( [UnknownRequest/Input], DeviceContent ) :-                     (i5)
   otherwise|
   storage_medium( Input?, DeviceContent ) .

lookup( Id, Term, Time, [(StoredId,Term,StoredTime)/DeviceContent] ) :-        (i6)
   less_or_equal( StoredTime, Time )| true.
lookup( Id, Term, Time, [NoMatch/DeviceContent] ) :-                           (i7)
   otherwise|                                % mismatch or Time < StoredTime
   lookup( Id, Term, Time, DeviceContent? ) .
```

Figure 10:
Program (i) – Specification of Extended Storage Medium

States of entities are stored as <ID,TERM,TIME> triples. The content of the disk is a list of such triples. The elements for a particular ID-TERM combination appear in temporal order. This ordering allows time to be included in an implementation's index calculation. Thus, the previous metalogical definition of query solution can be efficiently applied.

The first clause for *storage_medium* handles assertions for which the ID is not given. In response, a unique ID is allocated and a frozen representation of the new term is added to the storage device. The second clause deals with assertions for which ID is given. The client is asserting that a new term is to associated with a specific ID. Only if the ID is already recorded in the history is the new triple *(Id,Term,Time)* is added to the disk.

A *query* necessitates a scan of the disk content list, looking for a matching entry. A matching *(Id,Term,Time)* entry is one for which *Id* is that of the query and *Time* is less than or equal to the time specified in the query. Once a match is found, the stored term is melted and unified with the query term. Failure results if this unification fails or if no matching entry is found. Clause (i4) permits a client to use

the earlier form of query. The semantics of that request are unchanged. The request is transformed to the new form using the current time.

This disk interface is superior to the earlier one in several respects. Firstly, it enables simpler file servers to be constructed. They need only supply the naming facility, and, if required, caching or access control – histories being now retained by the storage medium. Less data needs to be saved by file servers to avoid loss of information on system failure. Secondly, garbage collection on the disk can be conducted. A *purge* directive is introduced which has the intuitive meaning "forget about knowledge asserted prior to time TIME which will no longer be referenced" (cf. purging old knowledge in Section 4.1.7.1). It is easy to add support for this directive to the storage medium.

```
storage_medium( [purge(PurgeTime)/Input], DeviceContent ) :-                    (i0)
   purge( Time, DeviceContent?, NewContent ),
   storage_medium( Input?, NewContent? ).

purge( PurgeTime, [], [] ) .                                                    (i8)
purge( PurgeTime, [(Id,Term,Time)/OldContents], [(Id,Term,Time)/NewContents] ) :-   (i9)
   less_or_equal( PurgeTime, Time )|          % retain this triple
   purge( PurgeTime, OldContents?, NewContents ) .
purge( PurgeTime, [(Id,Term,Time)/OldContents], NewContents ) :-                (i10)
   less( Time, PurgeTime )|                   % "old" knowledge – do not retain
   purge( PurgeTime, OldContents?, NewContents ) .
```

Figure 11:
Program (i') – Garbage Collection Extension to Program (i)

These clauses would be added to the earlier Program (i). They allow removal of triples relevant prior to the given time. The event calculus thus allows the modelling of time-dependent behavior within a disk.

A more selective *purge* could be defined. For example, *purge(Id, Time)* might remove only those triples having ID *Id* and time previous to *Time*. The extensions to Program (i) are straightforward.

In PARLOG, restricted unification is available [ClGr86]. '<=', for example, is a one-way unification primitive. It can only bind variables in its left argument. It suspends if it can only proceed by binding a variable in its right argument. If *storage_medium* (of Program (i)) is formulated in PARLOG, an interesting alternative is possible through the addition of the following integrity constraint. An asserted term must be `<=´-unifiable with all previously stored terms (prior to being frozen) for the same ID. That is, the term to be stored must be less general than earlier ones. Already-stored terms would be melted before performing the unification, and only one unification – with that most recently stored – would be necessary. Entity histories would then record successive approximations to a final ground

form. The resulting model has interesting parallels with Program (b), where successive approximations can also be obtained.

Models of secondary storage as a history of changes to entities are also found elsewhere. For example, a write-once laser disk file system developed at M.I.T. [Garf85] maintains, via a system of backward references, (all) past versions of files. In fact, the incorporation of laser disks into conventional operating systems often includes the ability to access the history of stored information [Vitt85]. Many of the techniques used in such file systems can be applied in logic systems. This is because the "write-once" nature of a laser disk maps nicely to the "instantiate-once" nature of logical variables. This is particularly true for languages like CP, as committed-choice nondeterminism stipulates that bindings once made cannot be undone.

4.1.7.3. Alternate Modes of Use

Clauses (i3), (i6), and (i7) force *query(Id,Term,Time)* be used only with a particular pattern of arguments. Specifically, processing suspends until both *Id* and *Time* are supplied. *Term* may be partially instantiated. It may be useful to modify Program (i) to allow different modes of *query*. For example, having *Term* completely instantiated but *Time* a variable would determine at what time entity *Id* had state *Term*. Similarly, if *Id* is unbound while *Term* and *Time* are bound, the ID of an entity having state *Term* at time *Time* would be found.

It is easy to add provision for these alternate modes of use in the specification, as they only require simple variations in the search of the disk content list. However, their efficient implementation may be more difficult. It is probable that an indexing mechanism designed to provide rapid access based on TIME would not be optimal for access based on ID. Further study is required to determine the utility of such alternate modes of use and their implementational cost.

4.1.8. Summary

This section has presented a technique for modelling declarative secondary storage devices. With the technique, a disk is treated as an independent entity possessing a knowledge base and simple inference mechanism. It uses these to process assertions and queries received from other components of the logic-based system. Two variant models were covered. Specifications for each were given. Implementation is the subject of the next chapter.

In an earlier discussion of these ideas [FoKu86], file servers were introduced. By introducing explicit time-stamps, file servers were generalized to maintain entity histories instead of single states. It was shown here that the storage system model can be extended so that the disk records entity histories, rather than simple facts. This formulation is more powerful and allows garbage collection of "old" knowledge. The ability to reclaim resources permits modelling of devices with bounded capacity.

4.2. Device Operation Through Function Computation

An alternate method for developing declarative, nonvolatile storage is possible. In this section, a model of I/O is achieved by extending in an intuitive and natural way the ideas in Chapters 2 and 3. I/O is viewed as the computation of members of a function. The resultant models are specified by logic programs describing device processes. It is later shown that this technique is easily adapted to interaction with other peripheral devices. Initially, discussion assumes infinite storage capacities. Later, finite capacity is considered.

4.2.1. Function Computation and Infinite Storage

As discussed in Section 4.1, a disk can be conceptualized at a high level, its contents being stored knowledge. In contrast, at a low level the information is just a collection of bytes. Interpretation at an intermediate level is also possible. Storage can be defined as a function (a restricted relation) from identifiers (e.g. symbolic or numeric location names) to terms:

disk_memory: IDENT $\rightarrow$ TERM

where the contents of a storage device define the function. IDENT and TERM are sets of ground terms. Typically, the cardinality of IDENT is less than that of TERM. Because disk_memory is a function, an identifier is associated with only one term. However, the function is many-to-one; that is, the same term can be associated with more than one identifier ("ID"). The function values never contain variables. Otherwise, it would indicate that an ID is associated with an infinite number of terms, a violation of a property of functions. For purposes of this discussion, the domain of disk_memory is restricted to the natural numbers.

In this model, the secondary storage device process computes the disk_memory function. The disk content is represented as a list in the DP's data state. Elements of the list record members of disk_memory which have already been computed. Known members are retained for efficiency (so they

don't have to be recomputed on queries) and to maintain the deterministic mapping from ID's to terms (so that an ID is never reported as associated with multiple terms). The disk_memory function is therefore represented at a metalevel.. <ID,Term> being a member of the list maintained by the storage medium DP means that <ID,Term> is a member of the function[12].

The device process can determine additional members of the function. These new members are incorporated into the device content by unification of variables. In committed-choice languages like CP, bindings once made cannot be "undone"; variables cannot be reused. To ensure that storage is always possible, the disk content is modelled as a list with an uninstantiated tail. This requires that the corresponding physical media be of (practically) infinite capacity. If such is not the case, at some point the disk becomes full and (henceforth) read-only (sections 4.2.1.1 and 4.2.3 address this topic further). Initially device content is '_' (an uninstantiated variable), and in general is

[(Id1, Term1), (Id2, Term2), . . . , (IdN, TermN)/Unallocated]

where each Id_i and $Term_i$ are ground. The variable *Unallocated* indicates that further members of the function exist, but have yet to be computed.

Storing information corresponds to calculating another member of the function and further instantiation of the tail of the device content list. That variable is instantiated to the (partial) list whose head is an ID-term pair (the information being added) and whose tail is a new variable (for subsequent addition of information). Storage of information thus involves unification of a logical variable and not destructive assignment. The term associated with a particular ID cannot be changed, as this would violate the definition of the function.

The remainder of the computer system can query the storage medium DP regarding members of the disk_memory function. There are two types of queries allowed: those giving an identifier and seeking the associated term (information retrieval), and those seeking the ID associated with a given term (information storage). These queries are captured by the requests *to_term(Id,Term)* and *to_id(Id,Term)*, respectively. Queries are satisfied by unification against elements of the device content list. This is similar to the style of interaction for the model in Section 4.1. However, it is based on a different con-

12. For convenience, we ignore the distinction between object- and and meta-level names. Since only ground terms are involved, an identity function could be used to map between them.

cept – calculation of members of a function, rather than maintenance of a clausal knowledge base – and exhibits different behavior. This is further discussed in Section 4.3.

To ensure that only ground terms are stored (as required by the model), the storage medium device process freezes all terms before storing them, and melts them on retrieval. This restriction has the benefit of not requiring an implementation to support shared variables on the disk.

If the ID for a certain term *term* has already been computed, the invariant that there can only be one function value for each ID must be maintained for subsequent *to_id(Id,term)* queries. This can be accomplished in two ways. Firstly, the process could search the device content list based on *term*, looking for a unifying pair. On success, the stored ID would be unified with *Id*. This is potentially very time-consuming and even impractical, especially if storage capacity is assumed to be boundless. However, the disk_memory function is many-to-one. Therefore, *term* could be associated with another ID; i.e. it could be related to another ID and (hence) stored again. This is preferable with an infinite capacity device, and is the approach used here.

Unlike the case above, the search for the term corresponding to a given ID is practical. In an implementation IDs can map easily (e.g. via a hash table or even directly) to the physical location of a stored term.

The program for the storage medium device process is now given. The DP determines the disk_memory function as it computes the relation which exists between a history (the stream of messages which is the DP's first argument) and a local database. The database is the content of the storage medium; i.e. a structure containing the already-computed members of the disk_memory function.

```
storage_medium( [to_id( Id, Term )/Input], DeviceContent, LastId ) :-          (j1)
    freeze( Term, FrozenTerm ),
    determine_unique_id( LastId, FrozenTerm?, Id ) |
    search( to_id, Id?, FrozenTerm?, DeviceContent ) &
    storage_medium( Input?, DeviceContent, Id? ).
storage_medium( [to_term(Id,Term)/Input], Content, LastId ) :-                 (j2)
    search( to_term, Id?, FrozenTerm, Content? ),
    melt( FrozenTerm?, Term ) |
    storage_medium( Input?, Content, LastId ).
storage_medium( [ProblemRequest/Input], Content, LastId ) :-                   (j3)
    otherwise |
    storage_medium( Input?, Content, LastId ).

search( _, Id, StoredTerm, [(Id,StoredTerm)/Content] ).                        (j4)
search( to_id, Id, StoredTerm, [NoMatch/Content] ) :-                          (j5)
    otherwise |
    search( to_id, Id, StoredTerm, Content ).
search( to_term, Id, StoredTerm, [NoMatch/Content] ) :-                        (j6)
    otherwise |
    search( to_term, Id, StoredTerm, Content? ).
```

Figure 12:
Program (j) – Storage Medium Specification (function computation approach)

Some explanation of the program is in order.

Clause (j1) processes *to_id(Id,Term)* (storage) queries. *Term* is frozen and a previously unused identifier calculated. This identifier is unified with *Id* in the message. *search* is then called with the new ID and the frozen term representation. Because the tail of the device content list is uninstantiated, the search will always succeed and bind the tail to *[(Id?,FrozenTerm?)/NewTail]*. This stores the information and generates a new uninstantiated tail. The processing of the message is now complete and *storage_medium* is called recursively with the remainder of the input stream, the (new) device content, and the freshly-determined identifier. Clause (j1) alone is responsible for adding new pairs to *DeviceContent*. Also, *search* is invoked with *Id* and *FrozenTerm* annotated as read-only. Thus, each element of every device content pair is a ground term or a read-only variable whose binding is being computed by a concurrent *determine_unique_id* or *freeze* computation[13]. Commitment is made to this clause after the *determine_unique_id* subgoal, as it is the only one which can fail (if the user supplies an incompatible ID).

13. In actuality, the presence of the commit before the *search* subgoal requires these two computations to have completed. However, in the general case, the commit would be replaced by a parallel AND, and the statement would still hold.

A *to_term* query initiates a search for a previously-computed member of the disk_memory function. The search is based on the given ID. If a binding is not given, the read-only annotation on the *Id* argument will cause the computation to suspend. This prevents the search from producing a binding for *Id*[14]. If and when the search succeeds, the (frozen) stored term is returned. This term, after being melted, must unify with *Term* given in the query. If these guard computations succeed, the query is answered, and commitment made. *storage_medium* then recurses to deal with further queries.

Clause (j3) prevents failure of the process. It is only a rudimentary facility. Additional capabilities are possible [Shap83a] and can even be added automatically [HiSS86]. Such modifications have been omitted for brevity and clarity.

determine_unique_ucr/3 need not be a complicated computation; it only needs to generate a unique ground identifier for any given values of its other two arguments. Since IDs are limited to positive integers (for purposes of discussion), this is easily done by a simple counting process. The definition could be

```
determine_unique_id( LastId, Term, IdToUse ) :-                    (j7)
    plus( LastId?, 1, IdToUse ) .
```

(assume *plus* is the obvious built-in predicate). Other algorithms are possible, providing a nonce identifier is generated each time.

The *freeze* and *melt* predicates are as described earlier (Sections 4.1.3 and 4.1.3.1).

The goal

search(Type, Id, Term, [(StoredId,StoredTerm)/Content])

succeeds via clause (j4) if *(Id,Term)* unifies with *(StoredId,StoredTerm)*. If the unification fails, clauses (j5) or (j6) continue the search with the remainder of the content. Clause (j5) is used if the search is to add the *(Id,Term)* pair to the list, should the end be encountered. In the recursive call in (j6), *Content* is annotated read-only. Therefore, it will suspend if the end of the list is encountered.

The semantics of the queries supported by the *storage_medium* device process are therefore as follows:

---

14. This is also made possible by the assumption that IDs are atomic (actually, numeric). The single read-only annotation is insufficient to enforce suspension in the case of structured terms.

to_id(Id,Term):

The storage medium DP determines an ID for *Term* and unifies *Id* with it; i.e. the process stores a ground representation of *Term*. *Id* is usually a variable. If given by the client, the binding for *Id* must agree with that determined by the storage medium (the client is trying the "guess" the function result). On success, *to_id* always results in a storage action, even if the same term has been stored previously.

to_term(Id,Term):

Id must be instantiated. The ID is used to locate the associated term stored on the storage medium. The term is then melted and unified with *Term*. *Term* is usually a variable. If given by the client, the binding for *Term* must unify with that determined by the storage medium.

If a *to_term* query is given for an ID which does not exist (has yet to be computed / stored), the search continues to the end of the content list and suspends until further elements are determined. This is different behavior from that when the ID exists, but the query term does not unify with the melted form of the stored term. In this latter case, the search succeeds, but the *melt* subgoal in clause (j2) fails. The query is thus discarded via clause (j3).

Because the commit operator in (j2) sequences processing of messages, a query with a (currently) nonexistent ID causes indefinite suspension. No further function members can be determined as no messages can be received until the guard computation (satisfying the offending query) terminates. This is an instance of the read/write race problem. The problem can be rectified by having *to_term* messages processed concurrently with acceptance of further messages by *storage_medium*; e.g.

```
storage_medium( [to_term(Id,Term)|Input], Content, LastId ) :-          (j2')
    determine_term( Id, Term, Content ),
    storage_medium( Input?, Content, LastId ).
determine_term( Id, Term, Content ) :-                                 (j8)
    search( to_term, Id?, FrozenTerm, Content? ),
    melt( FrozenTerm?, Term ) |
    true.
determine_term( Id, Term, Content ) :-                                 (j9)
    otherwise | true.
```

Such a modification does not affect the integrity of the device content, *Content*.

Data integrity of the device content list is maintained through the sequential-AND connective in clause (j1). It forces storing operations to complete before any further queries are processed, in

particular, more *to_id* messages. An alternative is to use dataflow synchronization. *search* would be redefined to have both input and output list arguments. As it stepped along its input list looking for the end, it would incrementally create the output list. Such an approach may be more elegant. It definitely increases the potential concurrency: a search could begin and proceed using the output list still being constructed as a result of a preceding query. Unfortunately, it requires decomposition and rebuilding of the device content data structure on each *to_id* query.

The search for the end of the device content list invoked in clause (j1) is inefficient. This is a disadvantage of the specification. However, because of the form of list elements and the manner in which *search* is called, no established portion of the list is modified (unification with already-stored pairs will either fail, or suspend and eventually fail). Therefore, an implementation of the model need not reflect the inefficiency of the search in the specification; the implementation could step directly to the list end. Also, a more efficient specification is possible using a difference list for the disk content. New pairs could then be added to the end in constant time.

4.2.1.1. Some Possibilities for Finite Capacity

It seems natural that finite storage can also be captured by Program (j). The significant changes apparently necessary are that the domain of the disk_memory function be a finite set and that *DeviceContent* be a fixed-length list

[(Id1,Term1),(Id2,Term2), . . . , (IdN,TermN)] .

If an element *(Id,Term)* of the list stores a member of the function, *Id* and *Term* are bound. Otherwise both are variable. *storage_medium* should also be extended with an argument recording the remaining (unallocated) capacity. The value would be checked and updated in clause (j1) on each *to_id* query. A query *to_term(Id,Term)* for which the *Id* is unknown would now fail (instead of suspending). One problem with this scheme is determining how many elements should initially exist in the device content list.

A slight variant has unallocated space on the disk represented by a list of *(Id,Term)* pairs:

[(Id1,Term1),(Id2,_), . . . , (IdN,_)] .

Each pair has a ground, preallocated ID as its first element. The second argument is either variable or instantiated. If variable, that ID is unallocated. Otherwise, the stored term is the second element. Since IDs are preallocated, the *determine_unique_id* subgoal in clause (j1) becomes unnecessary. This varia-

tion also requires use and maintenance of the remaining storage capacity value.

Unfortunately, both these approaches are lacking. After some finite number of transactions which store information, no further "free" pairs *(UcrI,TermI)* exist. From this point, the storage medium is strictly read-only. Finite storage has been captured, but not re-use. As shown shortly, there are superior approaches to capturing finite storage capacity.

4.2.1.2. Other Extensions and Restrictions

The requirement that *Id* in a *to_term(Id,Term)* query be ground makes a potential implementation simpler. An alternate scheme is to allow structured terms as IDs and permit *Id* to be partially instantiated. This alternative has not been investigated. It requires, at least, pattern-directed searches of a disk's content.

Suppose the list

[(1,term1),(2,term2), . . . , (n,termN)/More]

represents the current state of a disk's memory content. It might be proposed that deletion be modelled by simply removing an element from the head of the list. For example, removing *(1,term1)* leaves

[(2,term2), . . . , (n,termN)/More] .

This approach has pitfalls. The *storage_medium* process is computing the disk_memory function. After the removal above, however, the term corresponding to ID *1* cannot be determined. A *to_term(1,Term)* request would suspend indefinitely, even though that element was already computed. The element cannot be consistently recomputed as any recalculated result must agree exactly with the previous one. The removal would thus effectively eliminate the function member with ID *1*.

It is not difficult to specify a storage model which uses only a single type of query, but with arbitrary patterns of arguments. For example, modest changes to Program (j) yield:

```
storage_medium( [query( Id, Term )|Input], DeviceContent ) :-                (k1)
   determine_unique_id( Term, DeviceContent, NewId ),
   query( Id, Term, NewId?, DeviceContent ) |
   storage_medium( Input?, DeviceContent ).
storage_medium( [ProblemRequest|Input], Content ) :-                         (k2)
   otherwise | true.

query( NewId, TermToStore, NewId, [(NewId,FrozenTerm?)|FurtherContent] ) :-  (k3)
   freeze( TermToStore, FrozenTerm ).
query( QueryId, QueryTerm, NewId, [(QueryId,StoredTerm)|RemainingContent] ) :-  (k4)
   melt( StoredTerm?, QueryTerm ).
query( QueryId, QueryTerm, NewId, [StoredPair|RemainingContent] ) :-         (k5)
   otherwise |
   query( QueryId, QueryTerm, NewId, RemainingContent ).
```

Figure 13:
Program (k) – Storage Medium Specification (most general)

Such a model is very powerful. It is, in fact, too general and unrestricted. It captures more than what is practically implementable. For example, it requires pattern-directed searches.

Another alternative to the model of Section 4.2.1 is to add freshly determined members of the disk_memory function to the front, instead of the end, of the device content list. Then the data state of the storage medium DP would in general be

[(IdM,TermM), . . . , (Id2,Term2),(Id1,Term1)] .

After information *TermN* is incorporated, it would become

[(IdN,TermN),(IdM,TermM), . . . , (Id2,Term2),(Id1,Term1)] .

The uninstantiated tail is no longer required and the *storage_medium* specification would be simpler. Growth of the disk content list would be reflected in an implementation by writing information starting at the physical end of the disk, and working toward the beginning. Notwithstanding these advantages, the content of a disk is represented as a list with uninstantiated tail in the model. It is most natural to start allocation at the beginning of a memory, and to add new information after the used portion. Also, addition of stored information is more intuitively described by unification involving an unbound list tail.

4.2.2. Function Computation Applied to Other Peripheral Devices

Describing I/O as function computation is general and works well for peripheral devices other than secondary storage systems. This is substantiated in this section.

Output to a simple video display can be modelled as computation of the function

video_output : IDENT → TERM

where IDENT and TERM are as before. The display is represented by a device process, *display*, which computes this function. As with *storage_medium*, it consumes an input stream of client queries. However, since it is strictly an output device, it accepts only *to_id* messages. For each term sent to it, it determines the associated identifier. The term is converted to a sequence-of-characters representation and retained in the local argument of the *display* process, the device content list. It represents the information appearing on the display. The list has an uninstantiated tail (one can always output more information to a display). However, only a finite number of elements of the video_output function are retained because a set amount of information can be displayed at once. This capacity is recorded by another process argument. If an addition to the content would cause the capacity to be exceeded, elements from the front of the list can be eliminated. With the storage medium, this was not possible (see Section 4.2.1.2). However, since *to_term* queries are not being accepted, it can be safely done in this case.

Instead of a single list of characters, the device content may instead be a list of lists. Each component list contains the characters of one line on the display. A line to which more characters can be added might be represented as a list with an uninstantiated tail. Because a display has a finite vertical and horizontal dimension, all lists involved have a maximal size.

Input devices can also be captured by the function computation technique. Input-only devices are typically nondeterministic. Specification of nondeterministic input differs significantly from the deterministic input (and output) of a disk. A representative input device is a terminal keyboard. It is modelled as a device process, *keyboard*, computing the function

keyboard_input : IDENT → TERM

The DP consumes a stream of client queries. Queries are restricted to *to_term(Id,Term)* messages (it is a read-only device). A client provides *Id* and requires the device to respond with *Term*, such that *(Id,Term)* is a member of keyboard_input. Previously-determined members of the function need not be retained if a different ID be given in each request. Hence, for simplicity, it is required that IDs in successive queries be ordered. The requirement is easily enforced in the *keyboard* specification.

A logic program is defined over an infinite universe of terms. These terms are constructed from a

finite set of symbols. Correspondingly, the specification program for *keyboard* can be either infinite or finite. It is infinite if the units of input are arbitrary terms, even if restricted to constants and ground, structured terms. A specification program might contain clauses similar to (d2) and (d3) earlier. *search* would be defined differently, as follows:

search(to_term, Id, Term) :– possible_term(Term).

possible_term(ComplexTerm) :– possible_complex_term(ComplexTerm).
possible_term(Constant) :– possible_constant(Constant).

The definitions of *possible_constant* and *possible_complex_term* are infinite.

A finite specification is possible if use is made of "constructor" (system) predicates such as *name* or =.. . These are common in Prolog systems [ClMe81, StSh86]. However, these predicates are meta-logical and complicate the semantics of a specification. Alternatively, the model could restrict the units of input from *keyboard* to character constants. The "construction" of terms would then be the responsibility of the client. Since the set of possible characters is finite, a specification is finite. In this case, clauses similar to (d2) and (d3) would again be used. *search* would be solved using

search(to_term, Id, Term) :– possible_char(Term).

possible_char('a'). . . . *possible_char('z').*
possible_char('A'). . . . *possible_char('Z').*
possible_char('0'). . . . *possible_char('9').*
possible_char(' '). . . . *possible_char('~').*

(assuming an ASCII character set).

In any case, because the device being modelled is nondeterministic, the program must be nondeterministic. For example, in the segment above, it is indeterminate which of the facts for *possible_char* will be used to satisfy the *possible_char(Term)* subgoal. This reflects the impossibility of knowing what a human user's next input will be.

4.2.3. Finite Storage and Relation Computation

By necessity, all computer storage devices have finite capacity. Section 4.2.1.1 suggested that a finite storage device modelled by the function computation technique is unsatisfactory for practical use. As described in this section, this is remedied if the storage medium is conceptualized as computing a relation.

The storage device is now modelled as computing a relation, say disk_storage, between a finite set of identifiers (IDENT') and the set of possible terms (TERM). For a relation, as opposed to a function, an element of the domain can map to multiple elements of the range. Therefore, the term reported to be associated with an identifier at one time may be different from that disclosed later. Disk I/O is deterministic. Thus at any time, only one term is reported as associated with a particular ID, though the association reported can change over time (time is marked by some concrete events). A storage medium DP specification following these principles is now described.

The specification of the DP defines a perpetual process, *storage_medium*. As in Program (j), the *storage_medium* predicate names a relation between two objects: *Stream* and *Content*. *Stream* is a history of client messages. The list *Content* retains previously determined members of the disk_storage relation. This new *storage_medium* process supports *to_id* and *to_term* queries as before. To meet the conditions above, a new client request, *recompute(Id)*, is introduced. *(Id,Term)* is an element of the data structure *Content* if, and only if, a term *to_id(Id,Term)* appears in *Stream* and no message *recompute(Id)* appears subsequently. That is, the latest *to_id* query gives the (current) value to report as associated with the ID. If two queries *to_id(Id,Term1)* and *to_id(Id,Term2)* appear in *Stream* where *Term1* is different from from *Term2*, a message *recompute(Id)* must occur between them. Hence, only one entry per ID need be retained in the content list.

Over time, therefore, this device process retains different terms as associated with an ID. This corresponds to physically storing different information over time at the same disk location. Hence, reuse of memory is allowed. The content of memory is represented as a list of finite size without limiting the number of transactions that can be processed.

Since the storage medium being modelled is of finite capacity, the specification program must allocate and reuse memory space wisely. Also, elaborateness is not desirable. One possible approach uses a simplification of the exponential buddy system algorithm for conventional memory management [AhHU83]. This is a first-fit scheme where storage is allocated in sizes which are powers of 2. The algorithm is used in managing elements of the device content list. The device content is modified from Program (j), being a list of triples *(Id,Term,Magnitude)*. The list is fully predetermined to

[(Id1,Term1,Magn1), (Id2,Term2,Magn2), . . . , (Idn,TermN,MagnN)]

where each Id_i is ground (IDs are preallocated to triples). Every $Term_i$ is initially uninstantiated. Each $Magn_i$ is ground and gives the storage capacity associated with a triple (and ID). Values are integral powers of 2. In general, $Term_i$ is either ground (indicating that information is stored by this triple) or variable (denoting that the triple is available for storage).

A specification conforming to these principles is:

```
storage_medium( [to_id( Id, Term )|Input], Content ) :-                    (m1)
   freeze( Term, FrozenTerm, FrozenTermSize ),
   magnitude( FrozenTermSize?, Magnitude ),
   search( to_id, Id, FrozenTerm?, Magnitude?, Content, _ )|
   storage_medium( Input?, Content ).
storage_medium( [to_term( Id, Term )|Input], Content ) :-                  (m2)
   search( to_term, Id?, StoredTerm, Content, _ ),
   melt( StoredTerm?, Term )|
   storage_medium( Input?, Content ).
storage_medium( [recompute( Id )|Input], Content ) :-                      (m3)
   search( recompute, Id?, _, _, Content, NewContent )|
   storage_medium( Input?, NewContent? ).
storage_medium( [ProblemRequest|Input], Content ) :-                       (m4)
   otherwise | true.

search( recompute, Id, _, _, [(Id,StoredTerm,Magnitude)|RemainingContent],  (m5)
        [(Id,NewFreeSpace,Magnitude)|RemainingContent] ).
search( to_id, Id, FrozenTerm, Magnitude,                                  (m6)
        [(Id,FrozenTerm,Magnitude)|RemainingContent], _ ).
                                          % must be same magnitude –
                                          % don't waste smaller space
search( to_term, Id, StoredTerm, _, [(Id,StoredTerm,_)|RemainingContent], _ ).  (m7)
search( Type, Id, Term, Magnitude, [NoMatch|RemainingContent],             (m8)
        [NoMatch|NewContent] ) :-
   otherwise |
   search( Type, Id, Term, Magnitude, RemainingContent, NewContent ).
```

Figure 14:
Program (m) – Finite Storage Medium Specification

This program is similar to Program (j), as evident by comparing the groups of clauses: (j1), (j4), (j5) versus (m1), (m6), (m8); (j2), (j4), (j6) versus (m2), (m7), (m8); (j3) versus (m4); (j4) versus (m5), (m6), (m7); and (j5), (j6) versus (m6). The correspondence is deliberate and illustrates that finite storage requires only modest source-level modifications. Added clauses and predicate arguments are mostly for the new *recompute* query. For example, the last argument of *search* is used only for *recompute*. Much of the explanation of Program (j) continues to apply. The differences between Programs (j) and (m) are interesting, though sometimes subtle. These differences are now examined.

Clause (m1) is responsible for information storage. A *determine_unique_id* subgoal is not needed because IDs are preset in the device content list. If a term was already stored, Program (j) forced storage of a duplicate. This was to eliminate the need for a search. However, in clause (m1), a search for an available triple with correct storage magnitude is already necessary. A check against duplication adds little. Thus, the specification stipulates that if a term (in frozen form) is stored already, it may not be stored again. Also, the *search* subgoal in (j1) was not expected to fail (infinite capacity is available). Here, storage could fail, so the *search* subgoal in (m1) is within the guard.

Clauses for information retrieval – (m2), (m7), and (m8) – are very similar to those in Program (j). This is because the magnitude argument is ignored for retrieval. If the *Id* in a *to_term(Id,Term)* query does not occur in the device content list, the *Id* is illegal and the query fails. If a triple having the specified ID does exist, but the related term is currently uninstantiated (has not yet been stored), the device process will suspend. These actions are consistent with the idea of computing the disk_storage relation. If a particular ID is not a member of the finite domain, the query fails. If the ID is in the domain, but the associated term is as yet unknown, the computation will wait until it is known. Unfortunately, because of the manner in which *melt* is invoked in (m2) and the serialization of request processing in (m1), (m2), and (m3), the device process will suspend indefinitely in the latter case. Section 4.2.1 already described methods to improve concurrency among three analogous clauses, and allow a DP to continue execution despite a suspending query.

recompute(Id) messages are dealt with by clause (m3). The clause initiates a search based on *Id*. According to (m5) and (m8), when the search succeeds, a new device content list is returned. In it, the *Term* portion of the found triple is made an uninstantiated variable – the previously stored term is "forgotten". Multiple *recompute* messages can be issued for the same ID without intervening *to_id* queries: *recompute* is idempotent.

The object of the search in clauses (m1), (m6), and (m8) is an available triple with capacity of a particular magnitude. The predicate *magnitude(Size,Magnitude)* is true if the numbers *Size* and *Magnitude* satisfy the equation

$$Magnitude = \lceil \log_2 Size \rceil \ .$$

For a term of a particular size, *Magnitude* is the size of the smallest preset storage area guaranteed to

hold the term. *magnitude/2* can be defined using evaluable arithmetic predicates [ClMe81, StSh86], if they exist in the logic-programming system, or by a set of clauses of the form

magnitude(Size, 0) :– 0 < Size, Size =< 1 | true.
magnitude(Size, 1) :– 1 < Size, Size =< 2 | true.
magnitude(Size, 2) :– 2 < Size, Size =< 4 | true.
:

The set is finite, and defines *magnitude/2* for *Size* values smaller than the total storage capacity.

This memory allocation scheme is a simplification of the exponential buddy system. Specifically, when storage areas of a given magnitude are exhausted, there is no provision to "split" larger ones. The search for unused storage in clauses (m1), (m6), and (m8) considers only triples of the same magnitude as the term to be stored. A disadvantage of the scheme in general is that as magnitudes are all powers of two, at worst half the space on the disk may be wasted. Otherwise, the scheme is known to exhibit good performance [AhHU83]. Most importantly, it is computationally simple.

This specification is simplistic in that areas of predetermined size are assumed to have been set aside on the disk, each area having a preset ID. Finite storage can be achieved without such preallocation, where triples, up to the (remaining) capacity of the device, are allocated dynamically. Reclamation of fragmented space is even possible. However, such specifications are longer and more difficult to understand. Therefore, a simpler scheme is given here.

The presented model demonstrates that finite storage can be captured by a generalization of the function computation technique. Positive aspects of the specification are: it is a modest extension of Program (j), it is still concise, it does not involve complex computations, minimal amounts of information must be maintained to support reuse of resources, and maintenance of this bookkeeping information is straightforward. More sophisticated schemes could be used, of course.

4.2.4. Summary

This section has developed and specified models of peripheral devices using the simple idea of computing members of a function. For a disk with unbounded capacity, the function is from a set of identifiers (select ground terms) to the set of all ground terms. A device process, corresponding to the physical device, computes members of this function as it satisfies queries from other nodes, the clients. Queries may be of two types: giving an identifier and seeking the related term, or giving a term and

desiring the associated identifier. This technique was used primarily to develop a declarative storage medium. The resultant specification in CP is concise and does not resort to side effects. Application of the technique to other peripheral devices was demonstrated as well. It was also shown how the storage medium model can be reformulated in alternate ways, one of which captures finite storage.

4.3. Relationship Between the Two Approaches

Two general techniques for developing declarative secondary storage models have been presented in this chapter. One views I/O as assertions and queries of knowledge communicated between autonomous logic inference systems. The other views I/O as the computation of a function or relation. Both techniques are not elaborate and extend to other peripheral devices. Though they have similarities, they differ in interesting ways.

The *assert* and *query* messages of the first approach are similar to the *to_id* and *to_term* messages supported by the second. However, the independent logic system approach has storage corresponding to assertions, and retrieval corresponding to queries. With function computation, both storage and retrieval are captured by queries; the desired operation is characterized by what information is being sought (an ID or term) with the query.

A more significant difference is the way a retrieval based on a "currently" illegal ID is processed. The former scheme assumes that the data base contains all relevant knowledge when processing a query; the query is processed under the closed-world assumption. Therefore, such a query will fail. This is a more conventional action. With the other scheme, an ID not being among the currently cached members of a function does not mean that it is not an element of the domain. A subsequent request may cause the mapping for the ID to be determined. The world is "open". Hence, on an instantaneously illegal ID, the computation must suspend. This philosophy is consistent with dataflow architectures.

The UCR concept was introduced in Section 4.1. However, the concept is equally plausible with the models in Section 4.2. For storage as computation of the disk_memory function, the ID can again be considered a unique, compact representation ("UCR") of the term it maps to.

For both approaches, the specifications given in CP can be translated to FCP [Bloc84] and executed as "regular" computations under the Logix system [SHHS86]. In this way the correctness of the specification programs can be tested. The resulting *storage_medium* processes provide a logically

correct forms of declaratively disk I/O. However, they are volatile.

A list has been used to represent the device content in all the presented models. This is not required by the two techniques. For many devices, a list is a natural way to view the content of the device. Other data structures are not precluded, however.

Under both paradigms, devices do not provide input unless explicitly prompted, either by *query* or *to_term* messages. They therefore have similarities with processes which are **lazy** functions in an early version of PARLOG [ClGr83]. Lazy functions only "produce" values on demand.

4.4. Assumption of Infinite Storage Capacity

In the two major developments of secondary storage in this chapter, unlimited capacity is assumed. This is a reasonable assumption. Write-once laser disks are available with capacities large enough to be regarded, for practical purposes, as infinite. Conventional file systems for such hardware are often based on just such an assumption [Garf85, Vitt85]. Additionally, the write-once property of such media prevents an implementation (inadvertently) using destructive assignment.

4.5. Other Work

Disks are treated declaratively by other authors. In functional languages, disks and other external devices can be regarded as data streams [Hend82, Jone84b, Youn85]. The entity consumes a stream of messages, generating corresponding messages on on output stream as a result of computation (function evaluation). The set of files in the file store define a function from identifiers to functional terms. Logic variables do not exist in the functional paradigm. This has both advantages and disadvantages for secondary storage. Their lack necessitates tagged splits and merges. However, it also eliminates problems generated by shared variables. There is no need, for example, to introduce a freeze concept. The descriptions of functional storage systems do not mention running implementations (on prototype machines or machine emulators), however. In the logic-programming paradigm, Shapiro [Shap84a] views input and output devices as predicates that compute infinite sequences. However, actual detailed specification is not addressed. Peripheral devices as independent logic systems are not considered.

In the function computation storage scheme (Section 4.2), unsatisfied retrieval requests are deferred until the desired information is written. A generalization of this idea arises in data flow architectures [ArIa83] where writes are allowed to arrive after reads. Memory is tagged, indicating whether

its content has been written and a retrieval can be immediately satisfied. If the tag is not set, a read request is deferred and the location is marked indicating that a read is outstanding. When the corresponding write eventually occurs, the read is satisfied.

A description of the storage model in Section 4.1.1 has appeared earlier [FoKu86]. Here that model is presented again, but in modified form. Some aspects of the model are developed in more detail. The scheme for copying terms before storage (Section 4.1.3) and storage through function computation were not addressed in the earlier paper.

4.6. Concluding Remarks

This section has developed models for declarative I/O. Two general techniques, peripheral devices as autonomous logic systems and as computations of functions, were investigated. Secondary (disk) storage served as a representative example, though application of methods to other devices was shown. Disk storage of both infinite and finite capacity was considered. Possible extensions, such as storage histories, were discussed. Specifications of the models were given in CP. These are executable, and could directly provide an inefficient and volatile, though correct, implementation. An actual implementation of one model is the subject of the next chapter.

The goal of term-based declarative I/O is met. Units of data transfer are terms. Interaction with the peripheral device, and the action of the device, can all be described declaratively by logic programs. Unification plays a key role. All described schemes place major importance on identifiers. These ground terms can be thought of as unique, compact representations of the terms input or output.

Certain of the specifications in this section have appeared elsewhere [FoKu86], formulated in the language PARLOG. The logic programs given here, in CP, are typically more concise, as PARLOG does not support full unification.

This section has demonstrated the use of logic as a formalism for specifying the operation of (components of) a computer system.

5. From Specification to Implementation

In the previous chapter, various models of secondary storage were developed. Any of the storage medium device process specifications can be executed as given to yield an implementation. However, the data state of the resulting process (the disk content) will be volatile. A more realistic implementation must incorporate a nonvolatile storage medium. In this chapter, one specification, that in Section 4.1.3.1, is developed into a prototype secondary storage system, a working implementation. The implementation language is FCP.

In the assumed hardware architecture, a device processor is responsible for interfacing to a peripheral device. The processor's operation is describable by a logic program. The computation specified by this program is a device process. However, is it possible, given the specification for a device process, to achieve an implementation (e.g. a device processor) in a systematic fashion? The subject of this chapter is a method for doing so. The specific case of nonvolatile secondary storage is considered. As the original specification expressed I/O interactions declaratively, the result is an implementation of declarative I/O.

To demonstrate the method, Program (g) of Chapter 4 is subjected to a sequence of enhancements and transformations. Transformations used maintain equivalence of logic programs [HiSS86, TaSa83, TaSa84]. Extensions made modify the model in known, controlled ways. The result is a CP program which describes an extended, though otherwise equivalent, model. In this resultant specification, access to stored information is performed in a special manner which facilitates implementation. The CP program is then transformed to an equivalent in FCP [Bloc84]. In this final specification, the operations to access information can be captured by FCP emulator instructions. The necessary instructions are added to a FCP emulator and an actual working prototype is realized. Though the transformation techniques employed are not novel, this application of them is.

True CP machines do not (yet) exist. However, emulators for a restricted form of the language, FCP, are available [HoSh86]. Hence, the implementation is in FCP. FCP is suited for the purpose. Like CP, its constructs allow specification and synchronization of concurrent activity. CP programs can be systematically transformed to equivalent FCP forms. An efficient emulator [HoSh86] for the language exists, as well as a programming/user environment, Logix [SHHS86]. Facilities for user extensions and customizations of the emulator are provided.

The implementation is described in significantly more depth than (already) reported elsewhere [FoKu86, Kusa87]. The presentation begins with a review of the model to be implemented. Section 5.2 describes the evolution of the CP to a "final" form. Conversion of the specification to FCP is the subject of Section 5.3. The subsequent section discusses the final steps of the implementation, including introduction of new "system predicates". A "real" (nonprototypic) implementation is addressed in Section 5.5. Sections 5.6, 5.7, and 5.8 outline the application of the method to a storage medium following the function computation paradigm, to other devices, and to other languages, respectively. Work to which interesting parallels can be drawn is mentioned in Section 5.9. Finally, Section 5.10 summarizes and concludes the chapter.

Each transformation step or alteration is described. The discussions are brief but thorough. Obvious or trivial detail is not included. Readers not familiar with the techniques used are referred to cited papers for more explanation.

5.1. Storage Medium Device Model

The specification implemented is Program (g) of Section 4.1.3.1. The model is summarized as follows. A nonvolatile storage medium is an independent node in a logic-based system. It consists of a knowledge base and simple inference mechanism. Its interaction with other nodes is specified by Program (g). The storage system is able to process assertions and queries made by other nodes. These can refer only to facts defining a single predicate, *disk*. Assertions are of the form *assert(Id,Term)*, and queries, *query(Id,Term)*. These assert the fact *disk(Id,Term)*, and query whether the fact *disk(Id,Term)* holds, respectively. An ID (the value of an *Id* argument) is an identifier by which client nodes can reference a fact in subsequent queries. It serves as a key or index for rapid retrieval. An ID is typically much simpler (and "smaller") than the associated term. IDs are ground terms. In this work, non-zero integers are used. To avoid costly searches, information added to the database is not checked for duplication. Thus two assertions of the same fact result in the information being stored twice, with a different identifier each time. There is no provision for retraction of assertions.

Several properties of this model make it attractive to implement; for example, failure of queries with illegal IDs, requiring IDs to be ground before a query can be satisfied, and storage of information in frozen form. It was also the first model devised and the only one fully formulated when an implementa-

tion was first attempted.

The model specified by Program (g) assumes infinite storage capacity. Therefore, the implementation must employ a physical device whose capacity is "practically infinite"; that is, so large as to be regarded as unbounded considering the task at hand. A likely candidate is therefore a write-once optical disk. For this implementation, an optical disk was not available and a Winchester-technology magnetic disk used instead. This hardware was adequate for the prototype implementation as its capacity was never exceeded during testing or exercise.

5.1.1. A Modest Extension

Because this storage medium model, as implemented, later forms the basis for developing a file system (see Chapter 6), the device process is extended to support the query *last_id(LastId)*. The storage medium responds by binding *LastId* to the ID in the fact asserted (stored) most recently. This modification requires addition of a local argument *LastId* to the *storage_medium* predicate (as in Program (j) of Section 4.2.1) and inclusion of the clause

```
storage_medium( [last_id( LastId )/Input], Content, LastId ) :-                (g0)
   storage_medium( Input?, Content, LastId ).
```

The satisfaction of *last_id(LastId)* involves only *LastId*. Addition of the argument does not affect the semantics of the remainder of the specification. Clauses (g1)-(g3) become

```
storage_medium( [assert( Id, Term )/Input], DeviceContent, LastId ) :-         (g1')
   freeze( Term, FrozenTerm ),
   allocate_unique_id( FrozenTerm?, Id ) |
   storage_medium( Input?, [(Id?,FrozenTerm?)/DeviceContent], Id? ).
storage_medium( [query( Id, Term )/Input], DeviceContent, LastId ) :-          (g2')
   lookup( Id?, FrozenTerm, DeviceContent? ),
   melt( FrozenTerm?, Term ) |
   storage_medium( Input?, DeviceContent, LastId ).
storage_medium( [UnknownRequest/Input], DeviceContent, LastId ) :-             (g3')
   otherwise |
   storage_medium( Input?, DeviceContent, LastId ).
```

LastId is "write-protected" and a binding can only be produced by the *allocate_unique_id* goal in clause (g1). *LastId* can be used in the computation of new IDs as in Section 4.2.1. *allocate_unique_id/3* becomes a simple increment function. Clause (g1) is then changed to

```
storage_medium( [assert( Id, Term )/Input], DeviceContent, LastId ) :-          (g1'')
    freeze( Term, FrozenTerm ),
    allocate_unique_id( LastId?, FrozenTerm?, Id ) |
    storage_medium( Input?, [(Id?,FrozenTerm?)/DeviceContent], Id? ).
```

5.2. Transformation Within CP

Program (g) of Chapter 4, with the enhancements above, is subjected to a sequence of modifications and transformations. The modifications extend the semantics, but do not alter the basic characteristics of the model. The transformations preserve equivalence [HiSS86, TaSa83, TaSa84]. The objective is a CP program in which all access to stored information is preformed in a very restricted, localized manner. This specification is then translated to FCP. The following subsections described each transformation or modification step. Most of the transformations used, such as fold/unfold, are based on tranformation rules for functional languages [DaBu76].

5.2.1. Step (a)

Additions to the the device content data structure, *DeviceContent*, through goal reduction of *storage_medium* are eliminated. Such additions only occur in clause (g1''). This action is delegated to a new, auxiliary predicate *add/4*:

```
add( Id, Term, Content, [(Id,Term)/Content] ).                                  (g6)
```

Clause (g1'') becomes

```
storage_medium( [assert( Id, Term )/Input], DeviceContent, LastId ) :-          (g1^a)
    freeze( Term, FrozenTerm ),
    allocate_unique_id( LastId, FrozenTerm?, Id ) |
    add( Id?, FrozenTerm?, DeviceContent, NewDeviceContent ) &
    storage_medium( Input?, NewDeviceContent?, Id? ).
```

Resolution of an *add/4* goal via clause (g6) only involves head-call unifications. As used here, *add/4* always succeeds. Thus, the previous (g1'') can be obtained by an unfolding transformation [TaSa84].

The sequential-AND in (g1^a) guarantees that the recursive call is not made until *NewDeviceContent* is bound to *[(Id?,FrozenTerm?)/DeviceContent]*. In (g1''), this term is constructed on the recursive call. Thus data synchronization is maintained.

5.2.2. Step (b)

The predicate *freeze(Term,FrozenTerm)* is extended to *freeze(Term,FrozenTerm,Size)*. The latter names the relation "the frozen representation of *Term* is *FrozenTerm*, and that representation has 'size'

Size". *freeze/3* is true for exactly the same values of *Term* and *FrozenTerm* that *freeze/2* is. Only one *Size* value exists for each term *FrozenTerm*. The size reflects the resources (constants, list elements, integers, etc.) necessary for the representation[15]. *Size* is always an integer greater than zero.

The previous simple algorithm for generating IDs is modified. Using *freeze/3*, *allocate_unique_id* can now generate nonce identifiers by taking into account the last ID allocated and the size of the frozen term to be stored:

```
allocate_unique_id( LastId, TermSize, IdToUse ) :-                    (g7)
  plus( LastId, TermSize, IdToUse ) .
```

Clause (g1) is modified to

```
storage_medium( [assert( Id, Term )|Input], DeviceContent, LastId ) :-    (g1b)
  freeze( Term, FrozenTerm, FrozenTermSize ),
  allocate_unique_id( LastId, FrozenTermSize?, Id )|
  add( Id?, FrozenTerm?, DeviceContent, NewDeviceContent ) &
  storage_medium( Input?, NewDeviceContent?, Id? ).
```

No other changes are made. The resulting IDs still constitute an increasing sequence of non-negative integers.

The second argument of *storage_medium/3* can now be described more precisely: the argument *DeviceContent* is a list, where if *(Id1,FTerm1)* and *(Id2,FTerm2)* are successive elements of the list, then

Id2 = Id1 + size(FTerm2) .

size(FTerm2) is the size of the frozen term *FTerm2*.

It is obvious that messages which succeeded, failed, or suspended before will have the same outcome under this revised formulation of *storage_medium*. But, for the same sequence of requests, the bindings for ID variables will almost certainly be different. The precise semantics have changed, but the overall model has not.

5.2.3. Step (c)

To each request supported by the storage medium device process an argument is added. This argument, *Result*, is unified with either *true* or *false*, indicating the disposition of the request. Previously, requests which created subgoal failure were discarded by clause (a3), allowing *storage_medium* to con-

---

15. A precise meaning of "size" depends upon the format chosen for representing the frozen term. This is further discussed in Section 5.4.3.

tinue execution. Now the *Result* argument reflects this failure. Clause (g3) is necessary only when requests other than *assert*, *query*, and *last_id* are received.

This modification involves the addition of a second clause for every request in Program (g). Each new clause has an *otherwise* guard. It binds the *Result* argument of the request to *false* if the guard of the other, pre-existing candidate clause fails. To report success, the heads of (g0), (g1), and (g2) are extended to bind *Result* to *true*. The changes are exemplified by the clauses for a query:

```
storage_medium( [query( Id, Term, true )|Input], DeviceContent, LastId ) :-        (g2c)
    lookup( Id?, FrozenTerm, DeviceContent? ),
    melt( FrozenTerm?, Term )|
    storage_medium( Input?, DeviceContent, LastId ).
storage_medium( [query( Id, Term, false )|Input], Content, LastId ) :-             (g2.5)
    otherwise|
    storage_medium( Input?, Content, LastId ).
```

Definitions of *lookup* and other predicates are not affected, and the action of the program is otherwise unchanged.

For a given sequence of requests, the resultant device content list is as in the previous formulation. Any request which succeeded earlier also succeeds now (with *Result* bound to *true*). A failing request again does not generate any output bindings except, in this case, *Result* being bound to *false*. The relation between *Input*, *DeviceContent*, and *LastId* named by *storage_medium* is thus changed somewhat. The model is extended, with a small change in semantics.

5.2.4. Step (d)

The computations in the guard of clause (g2) are relocated to the clause body. Transformation of CP programs to FCP also involves the movement of goals from guard to body, so the technique developed for that purpose [Bloc84] can be employed. Clause (g2.5) is removed and (g2) is transformed to

```
storage_medium( [query( Id, Term, Result )|Input], DeviceContent, LastId ) :-     (g2d)
    lookup( Id?, FrozenTerm, DeviceContent?, LookupResult ),
    melt( LookupResult?, FrozenTerm?, Term, Result ),
    storage_medium_wait( Result?, Input?, DeviceContent, Id? ).
```

This necessitates definitions for *storage_medium_wait/4*, and *melt/4*, and the following modifications to *lookup/3*:

```
storage_medium_wait( Result, Input, DeviceContent, LastId ) :-                (g8)
   wait( Result ) |
   storage_medium( Input, DeviceContent, LastId ).

melt( false, FrozenTerm, Term, false ).                                       (g9)
melt( true, FrozenTerm, Term, true ) :-                                       (g10)
   melt( FrozenTerm, Term ) | true.
melt( true, FrozenTerm, Term, false ) :-                                      (g11)
   otherwise | true.

lookup( Id, Term, [(Id,Term)/DeviceContent], true ).                          (g4d)
lookup( Id, Term, [], false ).                                                (g4.5)
lookup( Id, Term, [NoMatch/DeviceContent], Result ) :-                        (g5d)
   otherwise |
   lookup( Id, Term, DeviceContent?, Result ).
```

storage_medium_wait simply suspends until its first argument – the result of the current request – is determined, and then invokes *storage_medium/3*. In *melt/4* and *lookup/4* the result of subgoal computation is treated explicitly, as an argument.

The added and modified clauses do not affect the input/output behaviour of *storage_medium*. The use of *storage_medium_wait* in (g2^d) enforces the same sequential processing of requests as achieved before by the commit operator. Equivalence is preserved.

5.2.5. Step (e)

Dataflow synchronization can enforce sequential computation [Kusa84b]. Thus, *storage_medium_wait/4* can be used to eliminate the sequential-AND in the most recent form of clause (g1). Goal-replacement transformation [TaSa84] yields:

```
storage_medium( [assert( Id, Term, Result )/Input], DeviceContent, LastId ) :-   (g1e)
   freeze( Term, FrozenTerm, FrozenTermSize ),
   allocate_unique_id( LastId, FrozenTermSize?, Id ) |
   add( Id?, FrozenTerm?, Content, NewContent, Result ),
   storage_medium_wait( Result?, Input?, NewContent?, Id? ).
```

The *add* predicate is extended with the argument upon which *storage_medium_wait/4* will delay. *add/4* always succeeded. Hence, the definition of *add/5* is:

```
add( Id, Term, Content, [(Id,Term)/Content], true ).                          (g6e)
```

The *Result* argument of an *assert* request is now bound to *true* when *add/5* in (g1^e) is resolved.

storage_medium_wait uses the metalogical predicate *wait/1* to achieve suspension. However, as all the computations involved are deterministic, semantics are preserved.

5.2.6. Step (f)

The *DeviceContent* argument of *storage_medium/3* is only (directly) manipulated by *lookup/3* and *add/4*. Hence, it is possible to define a process conjunctively parallel to *storage_medium* which maintains the list *DeviceContent*, and invokes *lookup* or *add* with that list. Further, the previous semantics of queries and assertions can be retained. The new process is called *data_type*, reflecting its close correspondence to an abstract data type.

The second argument of *storage_medium* is replaced by a stream to the *data_type* process. Invocations of *add* in clause (g1) and *lookup* in (g2) are replaced by subgoals which place messages on the stream. These messages result in the invocation of *add* or *lookup* within *data_type*. *data_type* has two arguments: an (input) stream for communications from *storage_medium* and the device content list. The following portions of Program (g) are thus altered:

```
storage_medium( [assert( Id, Term, Result )|Input], OperStrm, LastId ) :-          (g1f)
    freeze( Term, FrozenTerm, FrozenTermSize ),
    allocate_unique_id( LastId, FrozenTermSize?, Id )|
    send( put_term(Id?,FrozenTerm?,Result), OperStrm, NewOperStrm ),
    storage_medium_wait( Result?, Input?, NewOperStrm, Id? ).
storage_medium( [query(Id,Term,Result)|Input], OperStrm, LastId ) :-              (g2f)
    send( get_term(Id?,FrozenTerm,LookupResult), OperStrm, NewOperStrm ),
    melt( LookupResult?, FrozenTerm?, Term, Result ),
    storage_medium_wait( Result?, Input?, NewOperStrm, LastId ).

data_type( [put_term( Id, Term, Result )|Input], Content ) :-                      (g12)
    add( Id, Term, Content, NewContent, Result ) &
    data_type( Input?, NewContent? ).
data_type( [get_term( Id, Term, Result )|Input], Content ) :-                      (g13)
    lookup( Id, Term, Content?, Result ) &
    data_type( Input?, Content ).
```

In the remaining clauses for *storage_medium/3* and in the definition of *storage_medium_wait/4*, the argument *DeviceContent* is replaced by *OperStrm*. However, these clauses never access *DeviceContent*. For them, the argument remains a variable and any change is only in the written form of the clause. Clauses for all other predicates are unchanged.

Messages to *data_type* are sent in order corresponding to the receipt sequence of motivating *assert* and *query* requests. The use of sequential-AND in clauses (g12) and (g13) means that access to *DeviceContent* is sequential. Therefore, for a given sequence of assertions and queries, the sequence of lookups on, and instantiations to, *DeviceContent* is as before.

data_type executes concurrently with the storage medium DP. Section 4.1.3 gave a specification for the initialization of *storage_medium*. It is changed to:

```
storage_medium( [init(storage_medium)|Input] ) :-
   oracle( DeviceContent, LastId ),
   storage_medium( Input?, OperStrm?, LastId? ),
   data_type( OperStrm?, DeviceContent? ).
```

The technique employed in this step is that of replacing calls to a goal by the transmission of a message to a parallel process which performs the intended computation and returns a result. It has already been used successfully in Logix to implement "remote procedure calls" [HiSS86, SHHS86]. It is a form of goal-replacement transformation [TaSa84]. As used here, the technique yields the same semantics for *assert*, *query*, and *last_id* requests. Externally, the specification program behaves as before.

This alteration necessitated step (e). Without that previous step, clause (g2) could not have been transformed. The *send* predicate would otherwise have been in the guard, and the item sent would never be known outside the guard. Thus, the recipient would never receive it. (Such a problem is easily overlooked, even in published papers [Kusa84a]!)

5.2.7. Step (g)

Much of the synchronization in the specification program is achieved by read-only annotations. Synchronization is made more explicit by adding guard predicates to the definition of *data_type*:

```
data_type( [put_term( Id, Term, Result )|Input], Content ) :-                (g12g)
   wait( Id ), wait( Term ) |
   add( Id, Term, Content, NewContent, Result ) &
   data_type( Input?, NewContent? ).
data_type( [get_term( Id, Term, Result )|Input], Content ) :-                (g13g)
   wait( Id ), var( Term ) |
   lookup( Id, Term, Content?, Result ) &
   data_type( Input?, Content ).
```

Since the modifications are metalogical, they must be understood as a change in operational semantics [BoKo82] of *data_type*. The added tests always succeed. *wait/1* can never fail; it only suspends until its argument is instantiated. The *var/1* guard goal will succeed because its argument is guaranteed to be a variable in (g2). Thus, the outcome of each *data_type* computation stays the same; at worst it will be delayed. This may be perceived as a change of semantics if the client process employs metalogical constructs.

The client of *data_type* is, in this case, *storage_medium*. Hence the effect of the modifications can be examined in detail. *data_type* proceeds via clause (g12) only after the guard of (g1) has completed. However, that guard computation determines bindings for the variables *Id* and *Term* of (g12). Thus, the two *wait/1* predicates in clause (g12) succeed immediately, and their introduction has no effect on the semantics of *storage_medium*. Unfortunately, similar guarantees cannot always be given for the *wait/1* predicate in (g13). If the variable *Id* in (g13) is not instantiated and *Content* is not the empty list, the guard – and hence the entire *data_type* process – will suspend. These same circumstances result in suspension of *data_type* in the previous formulation because of suspending head-call unification in the resolution of *lookup/4*. A more interesting case is when *Content* is *[]* and *Id* is unbound. In the previous formulation, *lookup* – and hence *data_type* – succeeded immediately, with *Result* bound to *false*. Now the *data_type* computation suspends until *Id* is instantiated. However, whatever *Id* is eventually bound to, *Result* will be bound to *false*. Thus *storage_medium* will compute the same relations, and give the same results for requests. Only its timing behaviour has changed, and that only slightly.

5.2.8. Final Formulation

Collecting the most recent version of all clauses, reordering and relabelling them, yields the following specification for the final form of the model in CP:

```
storage_medium( [last_id( LastId, true )/Input], OperStrm, LastId ) :-                (n1)
   storage_medium( Input?, OperStrm, LastId ).
storage_medium( [last_id( _, false )/Input], OperStrm, LastId ) :-                    (n2)
   otherwise |
   storage_medium( Input?, OperStrm, LastId ).
storage_medium( [assert( Id, Term, Result )/Input], OperStrm, LastId ) :-             (n3)
   freeze( Term, FrozenTerm, FrozenTermSize ),
   allocate_unique_id( LastId, FrozenTermSize?, Id ) |
   send( put_term(Id?,FrozenTerm?,Result), OperStrm, NewOperStrm ),
   storage_medium_wait( Result?, Input?, NewOperStrm, Id? ).
storage_medium( [assert( Id, Term, false )/Input], OperStrm, LastId ) :-              (n4)
   otherwise |
   storage_medium( Input?, OperStrm, LastId ).
storage_medium( [query(Id,Term,Result)/Input], OperStrm, LastId ) :-                  (n5)
   send( get_term(Id?,FrozenTerm,LookupResult), OperStrm, NewOperStrm ),
   melt( LookupResult?, FrozenTerm?, Term, Result ),
   storage_medium_wait( Result?, Input?, NewOperStrm, LastId ).
storage_medium( [UnknownRequest/Input], OperStrm, LastId ) :-                         (n6)
   otherwise |
   storage_medium( Input?, OperStrm, LastId ).

storage_medium_wait( Result, Input, OperStrm, LastId ) :-                             (n7)
   wait( Result ) |
   storage_medium( Input?, OperStrm, LastId ).

lookup( Id, Term, [(Id,Term)/DeviceContent], true ).                                  (n8)
lookup( Id, Term, [], false ).                                                        (n9)
lookup( Id, Term, [NoMatch/DeviceContent], Result ) :-                                (n10)
   otherwise |
   lookup( Id, Term, DeviceContent?, Result ).

allocate_unique_id( LastId, TermSize, IdToUse ) :-                                    (n11)
   plus( LastId, TermSize, IdToUse ) .

add( Id, Term, Content, [(Id,Term)/Content], true ).                                  (n12)

melt( false, FrozenTerm, Term, false ).                                              (n13)
melt( true, FrozenTerm, Term, true ) :-                                               (n14)
   melt( FrozenTerm, Term ) | true.
melt( true, FrozenTerm, Term, false ) :-                                              (n15)
   otherwise | true.

data_type( [put_term( Id, Term, Result )/Input], Content ) :-                         (n16)
   wait( Id ), wait( Term ) |
   add( Id, Term, Content, NewContent, Result ) &
   data_type( Input?, NewContent? ).
data_type( [get_term( Id, Term, Result )/Input], Content ) :-                         (n17)
   wait( Id ), var( Term ) |
   lookup( Id, Term, Content?, Result ) &
   data_type( Input?, Content ).
```

Figure 15:
Program (n) – Final Transformed Storage Medium Specification in CP

This specification program must now be translated to FCP.

Unfortunately, this program is not particularly robust. For example, failure of the entire system might result from the client making a request in which the *Result* argument is bound to something other than the response that would otherwise be generated. Methods for handling such problems are known. They were not incorporated as they complicate the specification.

5.3. Translation to FCP

Program (n) specifies a declarative storage model in CP. It is an enhanced form of the model specified by Program (n). The next step towards implementation is translation to FCP.

Program (n) is deterministic. For deterministic programs, FCP differs from CP in that only system-defined test predicates are allowed as guard goals (CP allows arbitrary user-defined predicates in guards). Most of the clauses of the specification program are already valid in FCP. The exceptions can be transformed to equivalent clauses in FCP using established methods [Bloc84].

Clause (n3) is not a valid FCP clause. To make it so, *freeze/3* and *allocate_unique_id/3* must be placed in the clause body. For simplicity we assume that *freeze* always succeeds. This is reasonable, as all terms have a frozen representation, and the second and third arguments of *freeze* (the output arguments) are local variables. *allocate_unique_id* may, on the other hand, fail. This would occur if the user supplied an ID different from that determined by the predicate. Hence, *storage_medium* is extended to to deal with a possible discrepancy. A new predicate, *storage_medium_aux* is introduced to process a new message,

assert(UserSuppliedId,CalculatedId,FrozenTerm,Result) .

This message is generated internally; it is not available for clients. Clause (n3) becomes responsible only for freezing the term and having a new ID allocated:

```
storage_medium( [assert( Id, Term, Result )/Input], OperStrm, LastId ) :-          (n3')
    freeze( Term, FrozenTerm, FrozenTermSize ),
    allocate_unique_id( LastId, FrozenTermSize?, NewId ),
    storage_medium_aux( [assert(Id,NewId?,FrozenTerm?,Result)/Input], OperStrm, Id ).
```

The remainder of the processing of the assertion is performed by *storage_medium_aux*. Now that the third argument of *allocate_unique_id/3*, its "output" argument, is made a local variable, it will never fail. Hence, *allocate_unique_id* can be called in the body of (n3).

The definition of *storage_medium_aux* is

```
storage_medium_aux( [assert(IdToUse,IdToUse,FrozenTerm,Result)|Input], OperStrm,      (n18)
                    LastId ) :-
  send( put_term(IdToUse?,FrozenTerm?,Result), OperStrm, NewOperStrm ),
  storage_medium_wait( Result?, Input?, NewOperStrm, IdToUse? ).
storage_medium_aux( [assert(IdToUse,NewId,FrozenTerm,false)|Input], OperStrm,      (n19)
                    LastId ) :-
  otherwise |
  storage_medium( Input?, OperStrm, LastId ).
```

It completes the processing of an assertion. A message is sent to *data_type* if the freshly-allocated ID unifies with the argument given by the client. Otherwise, the *Result* argument of the request is unified with *false* and the request is discarded.

Clause (n4) is also affected by these changes. It becomes unnecessary and is removed.

Clause (n14) is also not valid in FCP; the *melt/2* subgoal must be moved to the body. To do so, the predicate is extended with a third argument, becoming *melt(FrozenTerm,Term,Result)*. It succeeds with *Result* bound to *true* whenever *melt(FrozenTerm,Term)* would succeed. If *melt/2* would fail, *melt/3* still succeeds, but with *Result* bound to *false* and *Term* not unified. *melt/3* is user-definable in FCP. However, the definition is similar to that for *melt/2*, and is omitted for brevity.

The new predicate *melt/4* makes clause (n15) unnecessary. It is therefore removed and the definition of *melt/4* becomes

```
melt( false, FrozenTerm, Term, false ).                                   (n13')
melt( true, FrozenTerm, Term, Result ) :-                                 (n14')
  melt( FrozenTerm, Term, Result ).
```

With these changes, Program (n) is converted to an FCP program.

5.3.1. Meta-Interpretation of *put_term* and *get_term*

The *data_type* process can be regarded as an interpreter capable of solving *get_term* and *put_term* goals. If the process is elevated to a meta-level computation, the subgoals

```
send( get_term(Id?,FrozenTerm,LookupResult), OperStrm, NewOperStrm )
send( put_term(Id?,FrozenTerm?,Result), OperStrm, NewOperStrm )
```

can be replaced by

```
get_term( Id?, FrozenTerm, LookupResult )
put_term( Id?, FrozenTerm?, Result ) ,
```

respectively. The replacement requires that the execution mechanism supporting the *storage_medium*

process communicate with *data_type*. When *put_term/3* or *get_term/3* goals are encountered, they are effectively passed to *data_type* for solution.

The FCP execution mechanism can be described by a meta-level interpreter. The "plain" one is [SaSh86]:

```
reduce( true ).
reduce( (A,B) ) :-
   reduce( A? ), reduce( B? ).
reduce( G ) :-
   G ≠ true, G ≠ (_,_) |
   clause( G, B ), reduce( B? ).
```

Figure 16:
Program (o) – Basic FCP Meta-Circular Interpreter

clause(A,Body) is a meta-logical predicate which encodes the clauses of the program. A clause of the form

$H :- G | B.$

is encoded as

$clause(H, B) :- G | true.$

The meta-interpreter (and hence the execution mechanism) is modified to support migration of *data_type* to the meta-level. A second argument is added to *reduce/1*. It is a stream variable for communications with *data_type*. Clauses are introduced to resolve *put_term* and *get_term* goals by placing an appropriate message on the stream. In effect, the goal is "forwarded" to *data_type*. The augmented meta-interpreter is:

```
reduce( true, [] ).
reduce( (A,B), Strm ) :-
   reduce( A?, StrmA ),
   reduce( B?, StrmB ),
   merge( StrmA?, StrmB? ).
reduce( get_term( Id, Term, Result ), [get_term( Id, Term, Result )] ).
reduce( put_term( Id, Term, Result ), [put_term( Id, Term, Result )] ).
reduce( G, Strm ) :-
   G ≠ true, G ≠ (_,_),
   G ≠ get_term(_,_,_), G ≠ put_term(_,_,_) |
   clause( G, B ), reduce( B?, Strm ).
```

Figure 17:
Program (p) – Augmented FCP Meta-Interpreter

merge/2 is a familiar CP and FCP predicate [Kusa84a, ShMi84, ShSa86] for merging two streams. It is described in Appendix A. The augmented meta-interpreter and *data_type* process execute concurrently.

Communications are initiated by coordinated access to a shared communication variable:

```
reduce( G ) :-
   reduce( G, Strm ),
   data_type( Strm?, [] ).
```

The *data_type* process receives and responds to messages serially. Thus, *data_type* behaves as before and its definition does not change. The process is simply elevated to a meta-level computation.

The only purpose for the second argument of *storage_medium* in Program (n) was to pass *put_term* and *get_term* messages to *data_type*. Since this has now been removed to the meta-level, the *OperStrm* argument can be eliminated from the *storage_medium*, *storage_medium_aux*, and *storage_medium_wait* predicates. The clauses (n5) and (n18), for instance, become

```
storage_medium( [query(Id,Term,Result)|Input], LastId ) :-                (n5')
   get_term( Id?, FrozenTerm, LookupResult ),
   melt( LookupResult?, FrozenTerm?, Term, Result ),
   storage_medium_wait( Result?, Input?, LastId ).
storage_medium_aux( [assert(Id,Id,FrozenTerm,Result)|Input], LastId ) :-  (n18')
   put_term( Id?, FrozenTerm?, Result ),
   storage_medium_wait( Result?, Input?, Id? ).
```

The other revised clauses appear in Program (q).

In Program (n), *get_term* and *put_term* messages are received by *data_type* in the same order that corresponding *query* and *assert* messages arrive for *storage_medium*. The continued use of *storage_medium_wait* preserves this ordering despite the move of *data_type* to the meta-level.

This interpretation of *data_type* and promotion of *get_term* and *put_term* messages to goals resembles the mechanism for implementing "remote procedure calls" (invocations of subgoals defined in another module) in FCP and Logix [SHHS86].

5.3.2. Final Formulation

After these transformations and modifications, the equivalent specification in FCP can now be given. Again clauses are relabelled.

```
storage_medium( [last_id( LastId, true )|Input], LastId ) :-                    (q1)
  storage_medium( Input?, LastId ).
storage_medium( [last_id( _, false )|Input], LastId ) :-                        (q2)
  otherwise |
  storage_medium( Input?, LastId ).
storage_medium( [assert( Id, Term, Result )|Input], LastId ) :-                 (q3)
  freeze( Term, FrozenTerm, FrozenTermSize ),
  allocate_unique_id( LastId, FrozenTermSize?, NewId ),
  storage_medium_aux( [assert(Id,NewId?,FrozenTerm?,Result)|Input], Id ).
storage_medium( [query(Id,Term,Result)|Input], LastId ) :-                      (q4)
  get_term( Id?, FrozenTerm, LookupResult ),
  melt( LookupResult?, FrozenTerm?, Term, Result ),
  storage_medium_wait( Result?, Input?, LastId ).
storage_medium( [UnknownRequest|Input], LastId ) :-                             (q5)
  otherwise |
  storage_medium( Input?, LastId ).

storage_medium_wait( Result, Input, LastId ) :-                                 (q6)
  wait( Result ) |
  storage_medium( Input?, LastId ).

storage_medium_aux( [assert(IdToUse,IdToUse,FrozenTerm,Result)|Input], LastId ) :-   (q7)
  put_term( IdToUse?, FrozenTerm?, Result ),
  storage_medium_wait( Result?, Input?, IdToUse? ).
storage_medium_aux( [assert(IdToUse,NewId,FrozenTerm,false)|Input], LastId ) :-      (q8)
  otherwise |
  storage_medium( Input?, LastId ).

lookup( Id, Term, [(Id,Term)|DeviceContent], true ).                            (q9)
lookup( Id, Term, [], false ).                                                  (q10)
lookup( Id, Term, [NoMatch|DeviceContent], Result ) :-                          (q11)
  otherwise |
  lookup( Id, Term, DeviceContent?, Result ).

allocate_unique_id( LastId, TermSize, IdToUse ) :-                              (q12)
  plus( LastId, TermSize, IdToUse ).

add( Id, Term, Content, [(Id,Term)|Content], true ).                            (q13)

melt( false, FrozenTerm, Term, false ).                                         (q14)
melt( true, FrozenTerm, Term, Result ) :-                                       (q15)
  melt( FrozenTerm, Term, Result ).
```

Figure 18:
Program (q) – Transformed Storage Medium Specification in FCP

It is assumed that the FCP execution mechanism specially recognizes the subgoals *get_term* and *put_term*. They are resolved by a (parallel) meta-level process, *data_type*. Though the *data_type* computation is moved to a meta-level, its definition remains as before:

```
data_type( [put_term( Id, Term, Result )/Input], Content ) :-                (q16)
    wait( Id ), wait( Term ) |
    add( Id, Term, Content, NewContent, Result ) &
    data_type( Input?, NewContent? ).
data_type( [get_term( Id, Term, Result )/Input], Content ) :-                (q17)
    wait( Id ), var( Term ) |
    lookup( Id, Term, Content?, Result ) &
    data_type( Input?, Content ).
```

5.3.3. Possible Further Modification

The number of clauses for processing *assert* requests has increased significantly since Program (h). However, the number could be reduced by a single change: making it a requirement that the *Id* argument in an assertion, *assert(Id,Term)*, be a variable. Such a restriction eliminates the possibility that *allocate_unique_id/3* fails, and clauses introduced to deal with this possibility become unnecessary. The change is most easily accomplished by a *var* test in clause (q3) and insertion of a clause to catch failure:

```
storage_medium( [assert( Id, Term, Result )/Input], LastId ) :-              (q3')
    var( Id ) |
    freeze( Term, FrozenTerm, FrozenTermSize ),
    allocate_unique_id( LastId, FrozenTermSize?, Id ),
    put_term( Id?, FrozenTerm?, Result ),
    storage_medium_wait( Result?, Input?, Id? )
storage_medium( [assert( Id, Term, false )/Input], LastId ) :-               (q3.5)
    otherwise |
    storage_medium( Input?, LastId ).
```

Clauses (q7) and (q8) can then be removed.

This modification alters the external semantics of the *storage_medium* process. While interesting, its further investigation is left for future work.

5.4. Implementing the Model

Program (q) specifies a model for secondary storage. It is (almost) a valid FCP program. Since a system for executing programs in FCP exists, a prototype storage medium device process can (potentially) be implemented. The only complication is the incorporation of the *data_type* process into the meta-level to deal with *put_term* and *get_term* goals.

The existing abstract machine emulator is an implementation of Program (o). Program (q) requires an emulator which is an implementation of Program (p) plus the *data_type* process. This can be provided by incorporating the additional functionality into the existing emulator and its support systems. That functionality is simply the ability to resolve the *put_term/3* and *get_term/3* goals. Thus, the

semantics of the two goals must be precisely determined.

5.4.1. Semantics of *put_term* and *get_term*

By examining the computation performed by *data_type*, it is possible to give detailed semantics for the *put_term* and *get_term* goals. They are:

put_term(Id, FrozenTerm, Result)

This goal always succeeds. Declaratively, the predicate names the relation "the term *(Id,FrozenTerm)* is the head of the device content list and *Result* is *true*". Operationally, the information *(Id,FrozenTerm)* is added to the disk content. *Id* is a new, unique identifier (as determined by the caller). *FrozenTerm* is a ground (enforced by the caller) representation of a term. *Result* is bound to *true*[16]. Satisfaction of the request suspends until both *Id* and *FrozenTerm* are (completely) instantiated.

get_term(Id, FrozenTerm, Result)

This goal always succeeds. Declaratively, the predicate names the relation "the term *(Id,FrozenTerm)* is an element of the device content list and *Result* is *true*, or the term is not an element of the list and *Result* is *false*". Operationally, a search is performed for the stored information associated with *Id*. Satisfaction of the request suspends until *Id* is instantiated. *FrozenTerm* must be a variable. The stored information, *(StoredId,StoredTerm)*, is unified with *(Id,FrozenTerm)*. More informally, the stored term associated with *Id* is retrieved and unified with *FrozenTerm*. If the term is found, *Result* is bound to *true*. If no information is stored associated with the given *Id*, *Result* is bound to *false*.

The operational semantics of the two predicates are, in fact, similar to the direct-access "reads" and "writes". However, they have declarative meaning. Also, the units of data transfer are logical (FCP) terms.

5.4.2. Extending the Abstract Machine Emulator

The Logix user environment facilitates the addition of new system predicates. Predicates corresponding to *put_term* and *get_term* are good candidates for addition as their semantics (given

16. The specification does not include a case where the operation fails and *Result* is bound to something else. This is discussed further Section 5.4.4.

above) are straightforward; they can be resolved directly by the emulator. Hence, the FCP implementation is extended to support *put_term/3* and *get_term/3* system predicates. As described in Appendix B, the FCP emulator and Logix support modules. Thus, the interface to these new predicates is encapsulated within a FCP module. The FCP abstract machine is therefore augmented by the functionality of the *data_type* process. The logical disk model can then be implemented.

To add the new system predicates, two new "guard kernel predicates" are introduced to the emulator's repertoire. Their functionality is precisely that of *put_term* and *get_term* described in Section 5.4.1. Their addition requires augmenting the emulator implementation program. An interface to the kernel predicates is provided by a FCP module defining a "monitor". This monitor fields *get_term/3* and *put_term/3* requests in the same manner as *data_type*. In response, it invokes the appropriate kernel predicate(s). The module, called *disk*, thus implements the new system predicates.

In FCP and Logix predicate names are not global, but local to the separately-compiled module in which the predicate is defined. A remote goal invocation feature is supported, however. Thus, predicates defined in other modules can be given as subgoals. (This feature is explained in Appendix B.) The following clause, which resembles (q4), is therefore syntactically valid:

```
storage_medium( [query( Id, Term, Result )|Input], LastId ) :-
    disk#get_term( Id?, FrozenTerm, LookUpResult ),
    melt( LookUpResult?, FrozenTerm?, Term, Result ),
    storage_medium_wait( Result?, Input?, LastId ).
```

When compiled, automatic source-to-source transformations change this to a clause almost identical to clause (n5) (e.g. arguments may be reordered). That is, the subgoal

```
disk#get_term( Id?, FrozenTerm, LookUpResult )
```

is transformed into the sending of a *get_term* request on a stream bound for *disk*. The mechanism assumes that the clauses to resolve *get_term* exist in the module *disk*. A remote procedure call of *disk#put_term/3* is treated analogously. Since *disk* has been implemented (above) to accept just such messages, the *get_term* and *put_term* goals are processed appropriately. Hence, *get_term/3* and *put_term/3* can be invoked in user-programs as (body) goals (remotely) defined in the module *disk*. The extended FCP execution mechanism has been achieved.

An actual working prototype has been realized following these steps [Kusa87]. The same model has been implemented in similar fashion on the Sequential PARLOG Machine [FoKu86].

The new system predicates *put_term/3* and *get_term/3* are no more complex than other system predicates supported by the FCP emulator [SHHS86], such as *get_file/3* or *put_module/2*. They are only slightly more complex than conventional direct-access "reads" and "writes".

In the logical specification, the disk content is local to the *data_type* process. In the implementation, the disk content is local to the emulator. Thus, the emulator is initiated with an (argument specifying an) area of memory to be used as the "physical" secondary storage. This is usually a file of the host operating system. This area is the object of *put_term* and *get_term* goals. The addresses of freshly allocated portions of this memory, "disk addresses", constitute an increasing sequence of integers. Hence, these disk addresses can be used as identifiers.

The semantics of the new system predicates correspond to those given in Section 5.4.1. However, some restatement and further explanation is in order. *put_term(Id,FrozenTerm,Result)* results in a frozen representation of a term, *FrozenTerm*, being stored with ID *Id*. *Id* is used directly as a disk address. The calling program guarantees that this ID has not been used before. This requires the predicate which determines the size of frozen representations (in this case *freeze*) to have knowledge of basic physical characteristics of the medium actually being used. It needs this to determine sizes in the "basic units" of the memory. *Result* indicates the result of the storage operation. It is typically bound to *true*, though Section 5.4.4 considers alternate values. Resolution of the goal suspends if, and while, the first two arguments are variable.

get_term(Id,FrozenTerm,Result) results in *FrozenTerm* being bound to the frozen term stored at disk address *Id*. If successful, *Result* is bound to *true*. If unsuccessful, or if *FrozenTerm* is not a variable upon call, *Result* is *false* and *FrozenTerm* is not bound. Resolution of the goal suspends if and while *Id* is uninstantiated.

Because of the characteristics of existing computer hardware, it is necessary to have, at some level, an imperative interface to the actual storage medium. This is hidden within the subprograms added to the FCP emulator. The predicates supported can still be understood declaratively.

5.4.3. The Implementation Program

Given that the functionality of the *data_type* process has been absorbed into the underlying computational engine, the storage medium implementation program is nothing more than Program (q), with

two minor extensions. These are the use of remote procedure calls in clauses (q4) and (q7). The clauses become:

```
storage_medium( [query(Id,Term,Result)|Input], LastId ) :-                (q4')
    disk#get_term( Id?, FrozenTerm, LookupResult ),
    melt( LookupResult?, FrozenTerm?, Term, Result ),
    storage_medium_wait( Result?, Input?, LastId ).
storage_medium_aux( [assert(Id,Id,FrozenTerm,Result)|Input], LastId ) :-  (q7')
    disk#put_term( Id?, FrozenTerm?, Result ),
    storage_medium_wait( Result?, Input?, Id? ).
```

disk is the model interfacing to the supplemental emulator capabilities.

As in specification program (n), the implemented *storage_medium* device process handles assertions and queries of the form *assert(Id,Term,Result)* and *query(Id,Term,Result)*. It also supports a request to know the last-allocated ID, *last_id(LastId,Result)*. The *Result* argument indicates the result (success or failure) of each request.

The *freeze* and *melt* predicates in Program (q) are user-definable in FCP. They were defined as user-programs in an initial implementation effort. However, for efficiency they were changed to system predicates in a later version. In either case, the format for frozen terms mimics that used by the emulator to store terms (data structures) in internal memory. The bytes making up the representation are contained in a **string**[17] to minimize space requirements.

Other formats for frozen term representation were certainly possible. The selected one offered definite benefits while incurring acceptable costs. The format makes the guard kernel predicates efficient, as a term is represented by a contiguous set of bytes. The bytes can be input/output directly from/to the physical medium – no manipulations or conversions are necessary. Because the format is close to internal representation, it makes melting – and hence queries – simpler and rapid. The cost is a somewhat slower *freeze* predicate and slightly larger representations. The inefficiency is in constructing the special format. It was assumed that queries (retrievals) are more frequent than assertions (storage operations) so should be made as efficient as possible, even at the sacrifice of some efficiency for assertions. Further technical details on *freeze*, *melt*, and the precise format of frozen terms are beyond the scope of this work.

---

17. In FCP, a string is a term composed of a contiguous sequence of numeric-valued bytes.

The size returned by the *freeze* predicate for a particular term reflects the number of basic units of storage medium resources (usually bytes) necessary to hold the representation. As disk addresses are IDs, and sizes of terms are in units of disk storage, the algorithm for generating new IDs (see clause (q12)) guarantees that terms are actually stored contiguously and sequentially on the physical medium.

The emulator addition supporting the new *put_term/3* predicate retains a cache of the last ID used (i.e. the last disk address successfully written to). This information is also recorded on the physical secondary storage medium. It is retained in case of reinitialization of the system (e.g. Logix). Otherwise, since the same information is maintained in the data state of *storage_medium*, it is redundant. A (further) extension was made to the emulator to retrieve this cached information. It is another guard kernel predicate, *oracle/2*. An interface to this predicate is provided by a module *oracle*. Initialization of the storage medium process is thus

```
storage_medium( [init( storage_medium )|Input] ) :-
  oracle#oracle( disk, LastId ),
  storage_medium( Input?, LastId? ).
```

The idea of an "oracle" for initializing the state of a device process was discussed in Chapter 3.

The specification programs such as (n) and (q) are not particularly robust so neither is the implementation. Fortunately, improvements to the specifications to make them more robust also carry over to the implementation.

5.4.4. Extensions and Further Work

Disk writes on a conventional system typically succeed. The usual reason for failure, when it does occur, is some capacity or quota being exceeded. Since infinite capacity is being assumed for the storage device, motivation for an assertion response other than *true* is rare. However, provision for errors on storage can be added to the specification in CP, Program (n), and the implementation. The definition of *add/5* in that program is extended with the clause

```
add( Id, Term, Content, Content, false ).
```

Now an *add* subgoal (in clause (n16)) can succeed in one of two ways, yielding a binding for *Result* of either *true* (indicating success) or *false* (indicating failure). Because of the OR-parallelism in CP, it is indeterminate which of the solutions is chosen. Practical operation would dictate that the clause yielding a "failure response" be chosen very rarely. However, OR-parallelism is missing from FCP. OR-

parallelism can be transformed to AND-parallelism [CoSh86], which FCP does support. Unfortunately, the transformation greatly complicates a program, for example the specification in FCP. The extension to have a response other than *true* generated is therefore possible, but is not pursued further at this time and left for future work.

Program (q) and its predecessors employ sequencing constructs (e.g. sequential-AND, *wait/1*, commit, read-only unification). Some of the sequential processing could be made parallel, not only to improve the potential efficiency of the specification and implementation, but also to make it more robust. For example, multiple queries (retrievals) could be processed concurrently while still preserving data integrity. Also, a user specifying a read-only variable for *Term* or a variable for *Id* in *query(Id,Term,Result)* can cause the DP to suspend indefinitely. Suspension could only affect that offending request if more parallelism were incorporated. Parallel constructs, if they are to be added, must be introduced in the original specification, Program (h), as they change the semantics of the model. Unfortunately, with greater parallelism comes much greater difficulty preserving semantics through the transformation steps. Data synchronization becomes a major issue that is not easily reconciled. This again is an avenue for further work.

5.5. A More Practical Implementation

The storage medium DP implementation is a successful prototype. It performs as required. It also suggests that a practical, working device whose external operations conform to the specified model is realizable. To illustrate, a "real" multi-processor logic inference machine is considered. An implementation might be achieved as follows. The "real" storage medium device (the device processor) is a processing node with a few minor enhancements. The processor interfaces to the others according to the set conventions, and supports the same (logic-programmed) machine language. However, it has attached to it physical storage devices. Also, its instruction set is extended to directly support the *put_term* and *get_term* system predicates (this may be added via microcode extension, for example). The node is otherwise indistinguishable from others. The processor would execute Program (q), or a counterpart. The result would be a declarative nonvolatile storage system.

The idea of a device processor is compatible with the evolution of microprocessors and the trend toward more "intelligent" hardware [Flei83]. This migration of intelligence is demonstrated by, for

example, ICOT's PSI where unification and many common operating system kernel functions are implemented in firmware [UYYT83, YYTN83].

5.6. Implementation Under Alternate Paradigm

The secondary storage model implemented in this chapter views I/O as assertions and queries of knowledge, communicated between autonomous logic inference systems. An alternate paradigm for modelling peripheral devices was also presented in Chapter 4: computation of functions. It is natural to ask, therefore, whether the model of secondary storage in Section 4.2 could be implemented by the same or similar strategy.

The storage medium device process specified by Program (k) of Section 4.2.1 shares many similarities with the DP implemented here. However, at least one difference between the two storage models may complicate implementation of Program (k). With Program (k), a query (retrieval) involves a search (see clause (k2)). The implementation method above dictates that the search functionality be moved into the execution mechanism with introduction of a new system system predicate. Program (k) stipulates that a retrieval based on an ID which has yet to be determined suspends – indefinitely. (In contrast, such a query would fail under Programs (h) or (n).) Therefore, execution of this new system predicate must suspend. Implementation of indefinite suspension is not difficult, but it is of questionable worth. Therefore, the specification may be modified as suggested in Section 4.2.1 to allow concurrent processing of queries. The suspending retrieval may then eventually be resumed, due to concurrent storage activities. A different type of suspension, which is more difficult to implement, would then be required of the system predicate. Each storage operation (which will be achieved by another introduced system predicate) must trigger a check of all suspended retrievals to see if the new ID is one being sought. Such a check, involving possibly many suspended queries, may prove inefficient. A form of "keyed wake-up" may be possible, but would require substantially more coding effort.

Apart from these complications, Program (k) appears conducive to implementation using the methodology discussed in this chapter. Pursuit of the implementation is a subject for further work.

5.7. Implementation of Other Devices

The implementation technique discussed is not specific to the storage medium; it is applicable to the specifications of other peripheral devices as well. Fold/unfold transformations are crucial to the

approach. However, Takeuchi [Take86] has shown that equivalence is not preserved for some nondeterministic FCP programs under this transformation. Until further work clarifies what nondeterministic programs are subject to the problem, it must be assumed that all are. Since specifications for some peripheral devices, such as keyboard input, are naturally nondeterministic, it is assumed, for now, that the implementation method is not applicable for them. This situation may change given more discriminating criteria for problematic programs, or alternate transformation schemes which preserve equivalence for a class of nondeterministic programs of which a (nondeterministic) device specification is a member.

Takeuchi's counter-example is dependent upon committed-choice nondeterminism. For deterministic FCP programs, fold/unfold transformations still hold. Devices other than secondary storage systems may have deterministic specifications. The techniques outlined in this chapter for realizing an implementation from a CP device specification are still applicable to them.

5.8. Implementation in Alternate Languages

As mentioned before, a specification for a declarative storage device has also been formulated in PARLOG. An implementation of it can also be achieved by the methodology above. The end result is a PARLOG program and an extension of the execution mechanism with capabilities similar to *put_term/3* and *get_term/3* above. Such an implementation for the SPM (Sequential PARLOG Machine) emulator[FGRS86] has already been realized and described [FoKu86]. The implementation is similar to that in FCP, with one important exception. A format for frozen terms which mimics the emulator's internal format is desirable to speed storage, retrieval, and melting operations. This requires an efficient data structure to hold contiguous sequences of arbitrary bytes. In FCP, strings were used. However, the PARLOG emulator currently lacks a suitable data structure. Consequently, in the PARLOG implementation the introduced system predicates incorporate the freeze and melt operations.

5.9. Other Work

Contemporaneous to this work, Kursawe [Kurs86] derived abstract machine instructions starting from a Prolog program. The original program is transformed and partial evaluated until an equivalent form is reached. In this form satisfaction of a goal is dependent on resolution of a set of special, simple predicates. The resolution of these predicates involves operations which can be performed directly by a machine. The predicates can then be "migrated" into the instruction set.

5.10. Summary and Conclusions

This chapter has described a method for implementing a nonvolatile secondary storage system. The result is a storage system prototype which is efficient, though nontrivial in functionality. It is achieved in a systematic fashion with emphasis on preservation of semantics at all steps. The prototype is practical as a basis for an "real" implementation. The methodology used is also applicable to other models of secondary storage, to other devices (at least those with deterministic specifications), and other committed-choice parallel logic-programming languages.

The implementation is summarized as follows. The device implemented was specified by Program (h) of Section 4.1.3.1. The specification is subjected to a series of transformations and extensions, arriving at an enhanced specification in CP, Program (n). The transformations used preserve equivalence. The extensions are restricted in effect, and retain the basic features of the model. Since a prototype system for developing and executing FCP programs exists, the enhanced specification is transformed to FCP. This is accomplished using recognized methods. In the FCP specification, a separate process is responsible for all updates to the device content list. This process is moved to the meta-level of the execution mechanism. Communications over streams from the remainder of the specification program to this process are transformed into object-level goals. An enhanced FCP meta-interpreter supports this further reformulation. The semantics of these special goals are straightforward, and the enhanced FCP execution mechanism can be directly implemented. The additional computational capability is provided by two new "system" predicates. The storage device implementation thus consists of a (modified) specification program in FCP and a modest extension of the abstract FCP machine.

Actual, working storage system prototypes have been constructed following these methods for both the FCP emulator and Sequential PARLOG Machine [FGRS86]. The implementations have been described in other works [FoKu86, Kusa87], though not the methodology used to obtain them.

Infinite storage capacity was assumed for the storage medium. This assumption is reasonable given the large – though finite – capacities of optical disks. Their write-once character is analogous to the single-assignment property of logic variables. Therefore, the operation of such devices is well adapted for logic programs.

The implemented device process provides useful, non-trivial functionality. It can form the basis

for realizing a file system. This is described in the next chapter.

A prototype implementation of declarative I/O has therefore been achieved for a non-trivial device. Logic, as realized by committed-choice logic-programming languages, has been used as a formalism and mechanism for specifying and implementing (portions of) a computer system. It is not presumed that all the details of a "practical" implementation have been solved. Rather, this thesis advocates that the first – and possibly most critical – step towards the development of any major computer system component is a clear, consistent, and powerful high-level model.

6. A File System Using Declarative Storage

In previous chapters, formal logic, as embodied by logic-programming languages, has been used to specify and implement components of a computer system. Secondary storage systems have served as representative examples. Chapter 5 described a method for implementing a declarative secondary storage medium. One way to test that component, and to show its power and utility, is to build a file system using its capabilities. Such is pursued in this chapter. An initial basic file system model is developed, specified, and implemented. An enhanced model is then derived and realized. The resultant designs are simple and powerful, though easy to implement. This is due to the expressiveness of concurrent logic-programming languages such as CP. The file systems exhibit interesting and innovative features. They also further demonstrate the feasibility and utility of term-based, declarative I/O.

"Practical" use of computers normally requires nonvolatile storage. In the development of logic-based computer systems then, the need for stable storage arises. Early forms of logic-inference machines are now being produced [NaNa87, YYTN83], but their file systems [HaYo84] depart little from convention; the novel properties of a logic-programming environment are not fully exploited. The file systems developed here, on the other hand, capitalize on these properties. So, for example, even though CP is sufficiently expressive, an analogue of a customary file system – say that of UNIX [RiTh74] – is not pursued.

A secondary storage system is a major facility in a computer system. Its development is typically a significant task. As mentioned earlier, its design and implementation are often indicative of the design and implementation of other portions of the computer system. A file system, as part of the secondary storage facilities, shares this importance and role. This chapter's development of a file system illustrates useful and novel programming techniques and constructs possible in CP, and other concurrent logic-programming languages, and their environments.

Term-based, declarative I/O was realized at a "low level" in the previous chapter. In this chapter declarative I/O is again sought, but at the higher level of a file system. An alternative is presented to the practise of having within a logic-programming environment only an interface to a traditional file system. Such an interface is usually imperative in nature. Further, a conceptual understanding of the facility often relies on destructive assignment and side-effects. The file system of a logic-based computer system should not rely on concepts which are contrary to the basis of logic-programming languages.

In any discussion of file storage, a number of fundamental questions must be addressed:

- What is the nature of the file abstraction?
- How is nonvolatility provided?
- How can file storage and retrieval be provided efficiently?

In addressing these questions, a number of important concepts arise, most notably those of unification-based information transfer, file system histories, unique compact representation, and lazy term expansion. As well, a file system becomes part of an interface between environments whose bounds are indicated by the scopes of variables.

A critical step towards development of any major computer system component – in this case, a file system – is a clean, consistent, and powerful high-level model. Therefore, an initial, basic model is given in Section 6.1. Side-effects and destructive assignment are avoided. In Section 6.2 it is shown how the model's specification, with little modification, forms an implementation. This design is described and critiqued. Using the knowledge thus gained an extended model can be formulated. This improved file system design is then presented in Section 6.3 together with its implementation. Features of the resultant file system are also discussed. Section 6.4 mentions related work. The last section summarizes the chapter, highlights important concepts, and proposes further work. Much of this material has already appeared [FoKu86, Kusa85b, Kusa87].

The proposed file system for a logic-based computer system is composed of a file system kernel and a set of servers. The kernel process is the lowest-level component and is discussed here at length. It maintains a file system as a database of <file name, file content> associations. Stable storage is achieved using checkpointing. Initially, the kernel is a single agent. It is eventually extended to a hierarchy of agents. Individual file servers are not described in detail; they are left for future works. However, examples of supplementary services they could provide are given, as well illustrations of how they might be realized. Overall, the file abstraction is built on the services of a secondary storage device process via user-level programs.

Logic variables are responsible for several atypical aspects of the file system design. Information transfer is unification-based, so file abstractions do not include explicit read and write constructs. The action of "closing" a file disappears – though it is partially replaced by user indication of when a file's

content is to be frozen (fixed). This is necessary due to practical considerations regarding logic variables [NaTU84].

Nonvolatile secondary storage is provided by a storage medium device process. Such a DP is described in Section 4.1.3 and implemented in Chapter 5. As in previous chapters, the storage medium is assumed to have infinite storage capacity. Files can be made nonvolatile by checkpointing thanks to this assumption.

The languages used are again Concurrent Prolog (CP) and Flat Concurrent Prolog (FCP). However, the presented ideas are also applicable to other parallel logic-programming languages [FoKu86], such as PARLOG. That is, no special properties of CP are utilized.

6.1. Simple Model

The fundamental purpose of any file system is providing other system software and user applications with the ability to store, retrieve, and update information in identifiable, nonvolatile units called files. Files are referenced through unique, human-readable keys called file names. File operations involve logical records defined at the user level.

A file server program was introduced in Section 4.1.7.1. It is a persistent process which merely maintains the ID of the stored file content as a local argument. This model was developed to motivate extending a storage medium device specification, and not for realizing a practical file system. Here, this last motivation is important. Thus, an alternate approach is taken.

A simplified form of the model, where the need for nonvolatility is ignored, is presented in this section. The file system kernel, file system servers, and possible variations are addressed. Stable storage is added in Section 6.2.

6.1.1. File System Kernel

The fundamental purpose of a file system is named, nonvolatile storage and retrieval of information. Such functionality is easily specified in a logic-based computer system by way of a perpetual process. The process maintains the information stored in the file system as its "data state" [Shap86b]. Client requests motivate the creation, removal, and access of files.

A perpetual process of this type is a **file system kernel process.** Its local argument is a database of

<file name, file content> associations. Individual associations (files) are affirmed (created), accessed, or revoked (removed) on receipt of appropriate messages. The result is a simple, though potent, file abstraction. The process can be implemented as a CP program.

In its most basic form, the kernel process is specified as

```
fs( [Req/Instrm], FileSysDB ) :-
    process_request( Req, FileSysDB, NewFileSysDB ),
    fs( Instrm?, NewFileSysDB? ).
```

where *process_request* represents the subgoals necessary for each file operation. The first argument of *fs/2* is a shared communication variable. Communications from clients are in the form of incomplete messages. The second argument is the file system data structure. A file is a pair, *(FName,FContent)*, of terms recorded in this data structure. *FName* is the file name. Using simple atoms as names provides a flat name space. Structured terms result in a hierarchical naming scheme. *FContent* is the file content and is an arbitrary term.

The file system kernel offers a nontrivial, straightforward, but easily-defined set of services. To create a new file a new association is "affirmed". The request *affirm(FName,FContent)* is sent to the process. Its intent is "the association between the name *FName* and the term *FContent* is affirmed". In response, the kernel process adds *(FName,FContent)* to its state. There is no need for messages or special operations to write into the file: the client simply instantiates *FContent*. Since the variable is shared by the kernel process, any instantiations made to it ("writes") are propagated to the process. Unification is responsible for the actual data transfer. The active participation of the file system kernel is not required.

The content of an existing file is "accessed". The request used is *access(FName,FContent)* . Again *FName* identifies the desired file. *FContent* is instantiated by the file system kernel to the current file content. Reading is then the examination of the term bound to *FContent*[18]. The file is modified by instantiating variables in the file content term. Read and write requests of the device process are unnecessary.

Associations (files) can also be "discarded" or "revoked". Hence, one last type of message is

18. Gregory [Greg85] observes that a file returned from the database is therefore analogous to a function which converges as it is being matched against.

needed: *revoke(FName, FContent)* . It requires that the kernel process query its database for a file designated by *FName*. The file content is unified with the second argument, *FContent*. The association of *FName* and *FContent* is subsequently removed from the process's internal state, as suggested by the predicate name. A typical use of such a request is removing a file: *revoke(FName,_)* .

A minimal file system kernel process with this functionality can be concisely specified:

```
fs( [affirm( FName, FContent )|ReqStrm], FileSysDB ) :-                    (r1)
   add_file( FName, FContent, FileSysDB, NewFileSysDB ),
   fs( ReqStrm?, NewFileSysDB? ).
fs( [access( FName, FContent )|ReqStrm], FileSysDB ) :-                    (r2)
   find_file( FName, FileSysDB, FContent, _ ),
   fs( ReqStrm?, FileSysDB ).
fs( [revoke( FName, FContent )|ReqStrm], FileSysDB ) :-                    (r3)
   find_file( FName, FileSysDB, FContent, NewFileSysDB ),
   fs( ReqStrm?, NewFileSysDB? ).

add_file( FName, FContent, FileSysDB, [(FName,FContent)|FileSysDB] ).      (r4)

find_file( FName, [(FName,FContent)|FileSysDB], FContent, FileSysDB ).     (r5)
find_file( FName, [F|FileSysDB], FContent, [F|NewFileSysDB] ) :-           (r6)
   otherwise |
   find_file( FName, FileSysDB?, FContent, NewFileSysDB ).
```

Figure 19:
Program (r) – Basic File System Kernel

For clarity, the file system data structure is portrayed as a simple list. Use of a more elaborate structure, such as a tree, requires modification of the predicates *find_file* and *add_file* only; the basic form of the program remains unchanged. The program is easily translated to other parallel logic-programming languages such as PARLOG (cf. the file store manager program of Clark and Gregory [ClGr84a]).

The program semantics are straightforward. The *fs* process is executed with two arguments, a communication stream (*ReqStrm*) and an internal database (*FileSysDB*). Execution suspends until a message is received. If *affirm(FName,FContent)* arrives, the pair *(FName,FContent)* is added to *FileSysDB* to form *NewFileSysDB*. On a *revoke* request, *find_file* is called with input arguments *FName* and *FileSysDB*. It succeeds with *FContent* bound to the appropriate file content, and *NewFileSysDB* bound to the database less the *(FName,FContent)* pair. For both requests the *fs* process recurses with the remainder of the input stream and *NewFileSysDB*. If *fs* receives an *access* request, it again invokes *find_file*, but retains the previous file data structure. Finally,

find_file(FName, FileSysDB, FContent, NewFileSysDB)

names the relation "the pair *(FName,FContent)* is a member of *FileSysDB*, and *NewFileSysDB* is *FileSysDB* with that pair removed".

With this view of file storage, a file is maintained as persistent information. It is logically "open" as long as its file content term is being shared by a client and the file system kernel. A file is "closed" when no part of the term is being shared. The kernel process is oblivious to a file being open or closed.

Several processes may access a file simultaneously. Multiple access is achieved by an existing client or the file system kernel sharing the file content term with other processes. If there are multiple attempts to write, all "writers" must agree on the binding(s) made, as with any case of concurrent processes instantiating the same shared variable [Shap83a]. Otherwise, unification will succeed for one "writer", and fail for the others[19].

New information may be written to a file content only if its content is not fully instantiated; i.e., if one of its constituent terms is a variable. For example, if the file content is the list

[t,h,i,s, ,f,i,l,e, ,h,a,s, ,s,e,t, ,c,o,n,t,e,n,t]

alteration is not possible. If it is instead

[a,p,p,e,n,d, ,t,o, ,t,h,i,s, ,f,i,l,e/X]

further bindings can be made. Modification of a file's content can only involve extension because variable bindings cannot be changed in a logic-programming language.

A characteristic of logic programs is that arguments are determined to be input or output according to their extent of instantiation at runtime. The situation is no different with *fs/2*. Thus, for example, an *access(FName,FContent)* request can be given where *FName* is variable and *FContent* is instantiated. This has a different effect from a normal *access* request: it determines if there is a file whose content unifies with *FContent*. If so, its name is bound to *FName*. Similarly, a request *revoke(_,_)* will remove the "first" file in the data structure, whatever its name.

Many capabilities are captured by Program (r). Unfortunately, *fs/2* is susceptible to failure. For example, it fails if an unrecognized request is received or a file to be accessed does not exist. Also, there is no check made for duplicate or uninstantiated names. These potential problems can be solved using

19. More precise semantics are dependent on the technique used to deal with simultaneous, multiple attempts to bind variables within the machine/implementation.

known techniques [Shap83a]. For example, to prevent failure on an unrecognized message, the following clause can be added:

```
fs( [Req|ReqStrm], FileSysDB ) :-
   otherwise | fs( ReqStrm?, FileSysDB ).
```

A practical enhancement of Program (r), especially in the case of error, is to include an explicit reply argument in messages. This requires only modest changes to *fs*. For example, the form of an *access* request can be extended to *access(FName,FContent,Reply)*, where *Reply* is bound by the kernel to either *true* or *false*. The modifications made to *fs/2* are:

```
fs( [access( FName, FContent, true )|ReqStrm], FileSysDB ) :-          (r2')
   fs_access( FName, FContent, Reply, FileSysDB ),
   fs( ReqStrm?, FileSysDB ).

fs_access( FName, FContent, true, FileSysDB ) :-                       (r7)
   find_file( FName, FileSysDB, FContent, _ ) |
   true.
fs_access( FName, FContent, false, FileSysDB ) :-                      (r8)
   otherwise | true.
```

Clauses for *affirm* and *recall* requests are changed similarly.

The *fs* process resembles a CP object [ShTa83], with *FileSysDB* corresponding to the internal state. However, it is not truly a CP object. An object's internal state can only be affected from the outside by sending a message to the object. This is the case with insertion and deletion of *(FName,FContent)* pairs. However, *FContent* terms are (meant to be) shared with client processes. The requirement is then violated, since instantiation of *FContent* changes the database of *(FName,FContent)* pairs, the internal state. Despite this, it is sometimes convenient and illustrative to treat *fs* as an object.

The presence of more than one file system kernel in a computer system poses no difficulties. CP processes are distinguishable by their communication channels. Clients specify a particular kernel by the channel selected for sending a (file system) request.

Direct implementation of this file system kernel as a device process is discussed in an earlier paper [Kusa85b]. The file system kernel is a generalization of a "file server" in a more conventional, distributed (networked) computer system.

6.1.2. File System Servers

The file system kernel process supports the basic services of creation, access, and removal (and eventually stable storage) of files. Servers are used to extend this functionality. Alternate file abstractions are possible, with varying file formats, naming schemes, access methods, and access restrictions. The nature of the file system perceived by a client then depends upon which server is employed. Users may introduce their own servers to add localized, custom features. This approach allows a great deal of flexibility. A multitude of differing servers and file abstractions are possible. This section discusses some examples.

Structured terms can be used as file names by the file system kernel to realize a hierarchical file naming scheme (UNIX pathnames [RiTh74] are one example). To alleviate the need to always specify complete hierarchical names, users can employ a server which automatically augments the name specified in each kernel request channeled through it. The server might take the following form:

```
file_server( [change_base_name( BaseName )|ReqStrm], OldBaseName, FsStrm ) :-
    file_server( ReqStrm?, BaseName?, FsStrm ).

file_server( [access( ShortName, FContent )|ReqStrm], BaseName, FsStrm ) :-
    construct_full_name( BaseName, ShortName?, FullName ),
    send( access( FullName?, FContent ), FsStrm, NewFsStrm ),
    file_server( ReqStrm?, BaseName, NewFsStrm ).
```

Figure 20:
Program (s) – Server to Aid With Hierarchical Names

The server process has three arguments: a channel for incoming messages, a retained "base name", and a channel to the kernel process. The base name is part of the process's internal state. For requests of the file system kernel – only the *access* case is shown above – the server constructs a full hierarchical name from the base name and the name in the message. It then forwards the modified request to the kernel. The base name is replaced on a *change_base_name* message. The server can easily be extended to have multiple retained base names and/or support multiple clients (users).

Since variables are single assignment in logic programs, it is not possible to update nonvariable portions of files. To update, the existing file must be discarded, and a new version created. However, a server can emulate a more conventional file abstraction, where updates are possible, and separate requests are necessary for opening, reading, writing, and closing a file. The file abstraction provided is a file as a sequence (list) of terms. Read/write operations are sequential and operate on single terms. The

following program is a simplified file server with this functionality:

```
file_server( [open( Name )|ReqStrm], _, _, _, FsStrm ) :-
    send( recall( Name, File ), FsStrm, NewFsStrm ),
    file_server( ReqStrm?, Name, File?, [], NewFsStrm ).

file_server( [read( Term )|ReqStrm], Name, [Term|RestOfFile], RevFile, FsStrm ) :-
    file_server( ReqStrm?, Name, RestOfFile?, [Term|RevFile], FsStrm ).

file_server( [write( Term )|ReqStrm], Name, [OldTerm|RestOfFile], RevFile, FsStrm ) :-
    file_server( ReqStrm?, Name, RestOfFile?, [Term|RevFile], FsStrm ).

file_server( [close|ReqStrm], Name, RestOfFile, RevFile, FsStrm ) :-
    append( RestOfFile?, RevFile, FinalRevFile ),
    reverse( FinalRevFile?, NewFile ),
    send( affirm( Name, NewFile? ), FsStrm, NewFsStrm ),
    file_server( ReqStrm?, _, [], [], NewFsStrm ).
```

Figure 21:
Program (t) – Server Providing Four Customary File Operations

The semantics of the program are straightforward. The arguments of *file_server/4* are, in order: a stream of incoming requests, the file name, the previous contents of the file, the revised contents, and a communication stream to the kernel process. The server must initially receive an *open* request. It recalls the desired file from the file system kernel and retains it as part of its internal state. *read* and *write* requests are processed with respect to this local information. A *read* causes the next term in the file to be returned to the client. The term is also added to the revised file content. On a *write* request, the next term is discarded, and the one supplied by the client is added to the revised file content. On receiving a *close*, the server appends the remaining original contents to the revised contents, reverses[20] the result to form the new file content, and affirms the revised form of the file.

The file system kernel allows file contents to be arbitrary terms, such as atoms, variables, lists, difference lists, and compound terms. A file system server can use these data structures to provide more commonplace file formats, such as character-stream or indexed files. Different formats can be supported simultaneously by different servers.

Other examples of facilities supplied by servers include data encryption/decryption and access restrictions. Servers may also be used to provide a consistent view of file storage given varying capabilities of multiple file system kernels.

---

20. Append operations could be performed in constant time, and the reversal eliminated, if a difference list was used to hold the revised contents.

6.1.3. Restrictions and Alternatives

As with any major computer system component, alternatives are possible in this file system model. Such alternatives are discussed here.

It might be required that a file content be a list of terms, instead of an arbitrary term. In the style of Program (t), the file system kernel could then support read and write requests for accessing the constituent terms of a file. However, this alternate approach still has the file system data structure as a process's internal state and data transfer through unification and shared variables. In fact, the alternative is essentially equivalent to the main approach because a list of terms is still a term, and an arbitrary term can be stored as a list of exactly one term.

The file system kernel does not support file updates, i.e. alterations of a portion of a file. This is a consequence of using a language in which variables are single-assignment. An update can still be achieved by obtaining a file's existing content, modifying or replacing the content, removing the old file, and creating a new one with the same name but new content. This is consistent with the behaviour of many conventional operating system utility programs which read a file, modify its content, and rewrite the file in its entirity (e.g. *ed* editor of UNIX).

The inability to update files need not be an encumbrance. Section 6.1.2 suggests one method to circumvent the limitation using a server. The file content becomes a list of terms. Another possibility is for the server to treat a file as the **history** of a list of terms, in the spirit of mutable arrays [ErRa84]. For example, the actual file content could be a list of lists, each logical data element having a separate list for values it has taken. On update, a new value is prepended to the beginning of the appropriate list(s). A simpler, less efficient scheme would have a file stored as a single list of *(ElementNumber,ElementValue)* pairs. Changes would be affected by prepending to the list. Unfortunately, a great many list elements might have to be examined to satisfy a read request.

As mentioned earlier, file access restrictions can be enforced by file servers. However, it may be advantageous to add such functionality directly to the file system kernel. It might be accomplished by way of "access keys". A file might then be a triple, *(FileName,AccessKey,FileContent)*. The key would be supplied by a client on affirming a file. The kernel would require clients to supply a "compatible" key on an *access* or *revoke* request.

Binding large data structures to variables and propagating such bindings are potential bottlenecks in a logic-programmed computer. Accordingly, files stored by the file system kernel should be kept small in size. However, very large files are commonplace on present-day computer systems. The observation that most large data collections have some form of internal organization (records, rasters, function or subroutine hierarchy, etc.) suggests a solution to this dilemma. Large collections of information can be stored in a number of smaller files, one logical unit per file[21]. A server can assist by accessing constituent files and emulating the larger view of the information. A user need not be aware of the actual granularity of storage. The server may be similar in style to Program (t), supporting *open*, *read*, *write*, and *close* requests.

It may prove necessary to impose size restrictions, both of file names and contents, in the file system. This is the result of a ubiquitous problem with CP, and logic-programming languages in general. In theory, it is always possible that some process will construct a term that exceeds the (necessarily finite) main memory capacity available to contain it. This is added motivation to have the file system kernel geared to small-sized files.

6.1.4. Summary

The file system is composed of a file system kernel and a collection of file system servers. The former provides the basic services of creation, access (reading or writing), removal, and (eventually) stable storage of files. It realizes an uncomplicated, though powerful model: a file store as a persistent process which maintains associations between names and values. A file is a pair, <file name, file content>, of terms. Clients gain access to a file by sharing the file content term with the file system kernel. Reading the file corresponds to examination of the term; writing, to instantiation. There is no need of explicit read or write operations, or of file closure. Information transfer does not require the active participation of the kernel process. File system servers enhance or modify this basic file abstraction. They can provide features of more conventional file systems, such as hierarchical directories or fixed, structured file formats. As shown in the next chapter, the simple model can be implemented by adding a checkpoint operation and the services of a storage medium device process.

---

21. Waterloo Port [MBSD83] and Waterloo UNIX Prolog [EmGo84] are examples of an operating system and a language subsystem, respectively, where storage of related information over a hierarchy of small files is the norm.

Further details, including discussion of implementing the file system kernel as a device process, are presented elsewhere [Kusa85b].

6.2. Implementation of Simple Model

Though the specification of a file system kernel process given in Program (r) is very short, the kernel's strength is substantial. File servers extend that strength. To study the feasibility of this approach to secondary storage, the file system kernel was implemented in FCP under the Logix user environment. No special hardware was used. This section describes the effort and presents conclusions based on the experience.

Program (r) can be executed as given to achieve a file system kernel. However, the file system data structure would then be volatile. A remaining concern in implementing the file system kernel then, is ensuring nonvolatility. Many algorithms are possible. However, an uncomplicated approach is taken here: the local argument of the process, the content of the file system, is checkpointed to a stable medium. The operation is not triggered automatically. Instead, a new client request, *checkpoint*, is supported. The nonvolatile storage facilities of a secondary storage device process, as implemented in the preceding chapter, are used. The file system kernel retains its declarative description.

To achieve the implementation, two modifications to Program (r) are necessary: a stream for messages to the storage medium is added as an argument of *fs*, and a clause is added to process *checkpoint* requests. (The only use of the new stream argument is in the new clause.) The opportunity is also taken to make the process more resilient. The resultant program (u) is given and explained below. It defines the perpetual process, *fs/3*, and is written in FCP.

```
fs( [affirm( FName, FContent, Response )|ReqStrm], FileSysDB, ArchiverStrm ) :-
   valid_file_name( FName?, ValidFName ),
   add_file( ValidFName?, FName?, FContent, FileSysDB, NewFileSysDB, Response ),
   fs( ReqStrm?, NewFileSysDB?, ArchiverStrm ).
fs( [access( FName, FContent, Response )|ReqStrm], FileSysDB, ArchiverStrm ) :-
   valid_file_name( FName?, ValidFName ),
   find_file( ValidFName?, FName?, FileSysDB, FContent, _, Response ),
   fs( ReqStrm?, FileSysDB, ArchiverStrm ).
fs( [revoke( FName, FContent, Response )|ReqStrm], FileSysDB, ArchiverStrm ) :-
   valid_file_name( FName?, ValidFName ),
   find_file( ValidFName?, FName?, FileSysDB, FContent, NewFileSysDB, Response ),
   fs( ReqStrm?, NewFileSysDB?, ArchiverStrm ).
fs( [checkpoint|ReqStrm], FileSysDB, [assert( _, FileSysDB, _ )|ArchiverStrm] ) :-
   wait( FileSysDB ) |                    % checkpoint file system content when available
   fs( ReqStrm?, FileSysDB, ArchiverStrm ).
fs( [], FileSysDB, [assert( _, FileSysDB, _ )] ).   % end of input stream: terminate gracefully
fs( [Req|ReqStrm], FileSysDB, ArchiverStrm ) :-
   otherwise |                            % ignore unknown requests
   fs( ReqStrm?, FileSysDB, ArchiverStrm ).

add_file( false, FName, FContent, FileSysDB, FileSysDB, false ).
                                          % file name not valid
add_file( true, FName, FContent, [], [(FName,FContent)], true ).
                                          % no duplication: add new file
add_file( true, FName, _, [(FName,FContent)|FileSysDB], [(FName,FContent)|FileSysDB], false ).
                                          % duplication: don't add new entry
add_file( true, FName, FContent, [File|FileSysDB], [File|NewFileSysDB], Response ) :-
   otherwise |                            % keep looking for duplicate entries
   add_file( true, FName, FContent, FileSysDB?, NewFileSysDB, Response ).

find_file( false, FName, FileSysDB, _, FileSysDB, false ).
                                          % file name not valid
find_file( true, FName, [], _, [], false ).   % end of files reached: search failed
find_file( true, FName, [(FName,FContent)|FileSysDB], FContent, FileSysDB, true ).
                                          % file found: search succeeded
find_file( true, FName, [File|FileSysDB], FContent, [File|NewFileSysDB], Response ) :-
   otherwise |                            % keep looking for file
   find_file( true, FName, FileSysDB?, FContent, NewFileSysDB, Response ).

valid_file_name( Name, Response ) :- ground( Name, Response ).
                                          % only ground terms are valid file names
```

Figure 22:
Program (u) – File System Kernel Process Implementation in FCP

Program (u) closely parallels Program (r). Comments explain differences whose motivations or effects may be less obvious. As FCP is less powerful than CP, the program is longer than an equivalent CP version. For example, the restriction to "simple guards" necessitates additional arguments and clauses for *add_file* and *find_file*.

The services provided by Program (u) differ from those of Program (r): file names must be ground, duplicate file names cannot exist, the search for a nonexistent file no longer causes process failure, and all requests include an additional *Response* argument. This argument is unified with either *true* or *false*, indicating whether the operation was valid and successful. These modifications were disussed earlier in the context of Program (r). They make the device process more robust, but do not significantly alter the model.

Nonvolatility is achieved through checkpointing and Program (u) supports a new client request, *checkpoint*. The operation is instigated explicitly by clients. A checkpoint is also automatically initiated by *fs* as part of its orderly termination (when the end of its input stream is detected). Actual archiving of information is performed by the storage medium device process, *storage_medium*.

The predicates *add_file* and *find_file* have changed from Program (r). For both, the first argument indicates whether the file name was found to be valid (by *valid_file_name/2*). *add_file* now checks for duplicate files. If a duplicate exists or the file name is invalid, a new entry is not added. The predicate's last argument indicates success or failure. *find_file* no longer fails if its search fails. Rather, its last argument indicates success or failure of the search.

A *valid_file_name* subgoal is added to the first three clauses for *fs/3*. Satisfaction of the client request suspends (at least) until *valid_file_name* binds its second argument, indicating the validity of the given file name. The criterion is simple: file names must be ground. Otherwise, the request is promptly failed. More restrictive or elaborate criteria are easily implemented. An interesting alternative is to suspend processing of a request until the file name argument is ground.

Two metalogical predicates are used whose definition does not appear. *wait/1* is a guard predicate which suspends until the principle functor of its argument is instantiated. *ground/2* is a predefined "library" predicate. *ground* unifies its second argument to *true* if its first argument is ground. Otherwise, its second argument is unified with *false*.

At initialization (see Chapter 3), *fs* and *storage_medium* are started as concurrent processes sharing a communication channel, *ArchiverStrm*. *fs* is the producer and *storage_medium*, the consumer. Initiation of the file system kernel takes advantage of the capability of the storage medium DP to report the ID of the last term stored. In this way, the state of the kernel can be restored after an inoperative period.

The kernel's initialization clause could be of the form

fs([init(fs)/ReqStrm], ArchiverStrm) :–
send(last_id(Id), ArchiverStrm, TempArchiverStrm),
send(query(Id?,PreviousFileSysDB, _), TempArchiverStrm, RemArchiverStrm),
fs(ReqStrm?, PreviousFileSysDB?, RemArchiverStrm).

Following initiation of *fs*, the only transmission between the two is *assert(_,FileSys)* which requests *storage_medium* to freeze the term *FileSys* and store it. The returned UCR is ignored.

6.2.1. Assessment of the Model and Implementation

This implementation realizes the model presented in Program (r), with slight extensions. It does so through a concise program. However, it has shortcomings:

(1) The data structure containing the file system content is repeatedly decomposed and rebuilt as it is traversed (by *add_file* and *find_file*) in handling requests. This consumes large amounts of main memory.

(2) A client must motivate the checkpoint operation. Not only is this tedious for clients, but it makes the timeliness of stored information unpredicatable without knowledge of user action.

(3) Storage granularity is too coarse. If a small change to one file is made, the entire file system data structure must be frozen and written to the storage medium.

(4) There are no provisions for quickly determining what files exist (e.g. a "catalogue"), only verification that a certain file is or is not present.

(5) The capacity of the file system is at most as large as the maximum-sized term that can be held internally by the inference machine. For realistic amounts of (file) information, this may be impractical.

Problem (4) is an easily-rectified limitation of the original model. The others are a consequence of the elementary implementation strategy used. For example, the coarse grain of storage is due to the broad scope of checkpoint operations. As will be shown, checkpointing is still a viable approach, providing it is done on a finer scale and the file abstraction is somewhat altered. The amount of information kept in the main memory of the inference machine can be lessened by application of the UCR concept.

6.2.2. Summary

Sections 6.1 and 6.2 have described a file system for a logic-based computer system. Many aspects of the design are noteworthy. The file system kernel facilitates the creation, access (reading or writing),

and removal of files. Through the cooperation of a secondary storage device process, it also provides for stable storage of information. A file is an entity maintained as part of the data state of a perpetual process. It can be manipulated like any term. File contents and file names are arbitrary terms, though names must be ground. A file is open when its content term is being shared between client and kernel. Unification supplants typical data transfer operations. Implementation of the kernel is obtained by simple extension of its specification in CP. Of primary concern is nonvolatile storage of the kernel's state. This is achieved handily by checkpointing. File system servers build on the capabilities of the file system kernel, and can provide useful features of conventional file systems.

Analysis of a naive implementation identified a number of problems. These are addressed in the next section. The file system model and its implementation are extended. It is shown that the basic model and implementation strategy remain viable.

6.3. An Extended File System

Previous sections described a simple file system model and its naive implementation. This section presents an enhanced file system, addressing shortcomings of the earlier version. The file abstraction and file system kernel are augmented. An implementation is smoothly obtained, again using the services of a secondary storage medium device process.

A file continues to be a <file name, file content> association. Each file name must still be a ground term. The file content may be an arbitrary term. Storage is again based on checkpointing. However, there is now a more definite sense of files being opened or closed. Clients indicate at what point information should be frozen (fixed) and stored. This makes checkpointing automatic and systematic. An indication to freeze a file's content signifies that the particular <file name, file content> association is completely elaborated and should be archived.

6.3.1. Requests

The following three requests are supported by the extended file system kernel:

affirm(FileName, FileContent, FreezeOn, Response),
access(FileName, FileContent, Response),
revoke(FileName, FileContent, Response).

A file is created (and opened) with an *affirm* request. The request has an extra argument which, when bound to a nonvariable, indicates that the file content should be frozen and recorded. This corresponds

to file closure. Creation of a file is not complete until it occurs. Subsequent instantiations of the file content term do not affect the stored information. With an *access* request, the file content is retrieved (and the file opened), but the stored information cannot be modified. The file content term can be further instantiated, though it will not be reflected in the state of the storage medium. The semantics of the *revoke* request are unchanged. The kernel no longer supports a *checkpoint* message.

The file system kernel accepts a new request,

inventory(FileNamePattern, ListOfFileNames) .

ListOfFileNames is bound to the list of all file names unifiable with *FileNamePattern*. For example, *inventory(_, ListOfFileNames)* binds *ListOfFileNames* to a list indicating all currently stored files.

The four requests supported by the extended kernel provide the basic services of a file system.

Modification of a file is achieved through a combination of existing functions. The previous content is *revoke*-ed, changes made, and the new content *affirm*-ed. To aid in this, two additional requests are supported by the file system kernel:

modify(FName, FContent, FreezeOn, Response) ,
replace(FName, FContent, FreezeOn, Response) .

The requests differ in whether the previous content is retained for modification (*modify*), or discarded (*replace*). However, they do not provide any new functionality. Internally they decompose to *revoke* and *affirm* requests.

6.3.2. File System History

The file system kernel process described in Section 6.1 maintained the current file system content as its data state. In this revised design, the process maintains the **history** of the file system content. Each entry in the history is a **version** of the file system content:

[FileSystemContentAtTimeM,
FileSystemContentAtTimeMminus1,
.
.
.
FileSystemContentAtTime0] .

Passage of time is marked by creations and removals of files; that is, a new version of the file system content exists after each file creation or removal. If stable storage is achieved by checkpointing, then (conceptually) the entire file system history must be checkpointed each time. This approach depends

heavily on the capabilities of the storage medium, most notably the interpretation of an ID as a unique, compact representation (UCR) of a term.

Initially, the file system history is

[FileSystemContentAtTime0] .

An individual file system content is a structure of *(FName,FContent)* pairs, as before. The history is recorded on the storage medium and has associated ID *UCR0*, say. When the file system is modified, a new file system content, *FileSystemContentAtTime1*, exists and the history must be updated. Conceptually, the file system history is then

[FileSystemContentAtTime1,
FileSystemContentAtTime0] . (α)

However, since *UCR0* is a compact representation of the earlier history, this term can be represented as

[FileSystemContentAtTime1 /UCR0] . (β)

Checkpointing this to the storage medium conceptually stores the entire history. A unique ID, say *UCR1*, is returned. It represents the term designated (β), which in turn represents the history (α). In general, the entire history of the file system at time *i* is represented by an ID UCR_i. The corresponding term retrievable from the storage medium is

$[\ FileSystemContent_i/UCR_{i-1}\ ]$.

In this way, UCRs make storage of large data structures more efficient. Further, *query* requests of the storage medium provide **lazy term expansion.** For example, in determining the history associated with UCR_i, each query will reveal another element in the history, along with the UCR of the remainder. The complete term is expanded gradually, subterms being examined only as required.

When the file system kernel is restarted (after some inoperative period), the preexisting file system history is necessary. This can be obtained via a *last_id* request of the storage medium[22]. Without this request, the file system could not span periods when the logic-inference machine or file system kernel

22. *last_id* may not provide the ID of the file system history if it was the case that operations were disrupted after a new file content was recorded (see ahead), but before a new file system history could be archived. Solutions to the problem exist. Some involve a distinction between stored history and file content terms. An encapsulating distinguishing function suffices. Another solution is to record with each file content term the UCR of the most recently archived file system history. Then if a *last_id* request after system restart gives the ID of a file content term, the correct history can still be obtained. Yet other solutions are possible, involving modifications to the file system kernel, as well as the storage medium DP and its implementation.

were inoperative.

6.3.3. File System Content

Each file system content version is a data structure composed of *(FName,FContent)* pairs. Without loss of generality, the data structure can again be a list; i.e.

$$\begin{aligned} CurrentFileSystemContent = [&(FName1,FContent1),\\ &(FName2,FContent2),\\ &\cdots\\ &(FNameN,FContentN)] \end{aligned}.$$

The UCR concept is also used in storage of individual files. When each *FContent* term is saved on the storage medium, an ID is returned. As each ID is a UCR, the above file system content is represented by

$$\begin{aligned} CurrentFileSystemContent = [&(FName1,FContentUcr1),\\ &(FName2,FContentUcr2),\\ &\cdots\\ &(FNameN,FContentUcrN)] \end{aligned}.$$

In each *(FName,FContentUcr)* pair, *FContentUcr* is the ID for the file content. The replacement of terms by UCRs diminishes the size of the list. The file server can selectively expand elements of the list – files – by querying the storage medium for the term corresponding to a UCR. Thus, lazy expansion also applies to the examination of the files.

If a particular file content reappears without modification in successive versions of file system content (the normal situation), resource wastage might be expected. However, the use of UCRs minimizes the problem. The UCR of the content is stored repeatedly, rather than the actual term. The UCR is in general much simpler and "smaller", and requires fewer resources.

6.3.4. Example

An example illustrates the previous points. Assume that at time *j* the file system history is represented by *UCRp* and the current file content is empty, *[]*. Consider addition of the file *example1* with content *data1*. The content term is stored, yielding ID *UCRa*. To store (a representation of) the new history, the term

$$[\,[(example1,UCRa)]\,|UCRp\,]$$

is constructed and saved. The storage medium returns an ID, say *UCRj*.

Suppose the file *example2* with content *number(2)* is now created. *number(2)* is recorded yielding an ID, *UCRb*. The new file system content is

[(example1,data1),(example2,number(2))] .

The file system history, now represented as

[[(example1,UCRa),(example2,UCRb))] /UCRj] ,

is stored. Say the ID *UCRk* is returned.

Assume the file *example1* is subsequently deleted. The pair representing the file is removed from the file system content; that is, the list of pairs becomes

[(example2,number(2))] .

The history that is stored is then

[[(example2,UCRb)]/UCRk] .

Conceptually, this last term represents the history of the file system content:

[[(example2,number(2))],
[(example1,data1),(example2,number(2))],
[(example1,data1)]
/UCRi] .

The leading three elements are lists representing, in order, the file system content after each of the mentioned creations or removals. This demonstrates how the UCR concept reduces the amount of information that must be stored.

6.3.5. Implementation

A specification of the enhanced file system kernel has been formulated in CP. It is of modest length. To implement the kernel, the clauses were translated to FCP. The resulting program executes within the Logix environment [SHHS86]. To exploit potential parallelism and hence improve efficiency, it implements the file system kernel as two modules. One module is simply an interface. It ensures that client requests are of proper form and that file names are ground. The other module is responsible for all file management. A concise form of the program appears in Appendix C. It uses the remote procedure call facility of Logix [SHHS86]. That is, a subgoal

Module#Subgoal

requires that *Subgoal* be resolved with respect to the predicate definitions given in module *Module*. This

feature, and modules, are discussed in Appendix B.

The fundamental data structure in the file system kernel is the file system history. However, this term is typically too large to fit into main memory. Due to the UCR and lazy term expansion concepts, only the ID of the most recently stored history and the current file system content need be retained as local data. UCRs are stored in the file content list instead of full file content terms. From the data state, any file system content version or file content can be selectively obtained.

To test the file system, the existing file services provided by the *file* module of Logix were replaced by those of the file system kernel. The experimental system was then used for "normal" FCP exercise and development work (including further work on the file system!). The file system performed as specified.

6.3.6. Further Extensions

Extensions to the file system are possible. Some are presented here. One has been implemented, and is described in depth.

6.3.6.1. Hierarchical Directories

Use of structured terms as file names provides a hierarchical name space. A "directory structure" results when names are restricted to atoms or functional terms of one argument, where the argument satisfies the same restriction (recursively). For example, files with names *down(a)* and *down(b)* are both in the directory named *down*. An equivalent scheme is achieved by restricting file names to lists of atoms.

Hierarchical naming can be supported in one of two ways:

- by a file server (the server mapping the names to a flat name space provided by the file system kernel); or
- implemented directly within the file system kernel.

Directory-structured naming capabilities were added to the file system using the latter approach. The file system requests in Section 6.3.1 are supported and many of the techniques already described were employed. For example, the file system history is still updated and checkpointed every time a change is made to the file system. Again, the UCR concept lessens the amount of information that is stored.

Up to now, the file system content has been a list of pairs, as in

CurrentFileSystemContent = [(FName1,FContent1),
(FName2,FContent2),
. . .
(FNameN,FContentN)]

This data structure is generalized, allowing elements of the content list to themselves be lists of files. The file system content becomes a list of terms of the form *f(FName,FContent)* and *d(DirName,DirContent)*. Files are represented by functional terms *f(FName,FContent)* instead of by pairs *(FName,FContent)*. *d(DirName,DirContent)* terms represent directories. *DirName* must be ground and is the directory name. *DirContent* is the directory content and can be any (valid) file system content; i.e., a list of *f(FName,FContent)* and *d(DirName,DirContent)* terms.

For example, if the file system contains files *down(a)*, *down(b)*, and *up* with content *file_content_1*, *file_content_2*, and *file_content_3*, respectively, the file content term is

[d(down, [f(a, file_content_1), f(b, file_content_2)]),
f(up, file_content_3)] .

The file system content of directory *down* is the list of two files *[f(a,file_content_1),f(b,file_content_2)]*.

Just as UCRs reduce storage costs of file content terms, so too for the directory content terms. For example, the file system content above may be represented by *[d(down,UCRd),f(up,UCR3)]*, where *UCRd* expands to *[f(a,UCR1),f(b,UCR2)]*. *UCR3* would expand to *file_content_3*.

The earlier file system kernel (of Section 6.3.5) is extended to a multiple-agent configuration to support this directory scheme. A "master" agent manages files in the root of the directory hierarchy. Subdirectories are served by "slave" agents. The agents exist in a tree defined by the directory structure. Edges of the tree are interserver communication paths (streams). Each slave has both an input stream from, and an output stream to, the agent responsible for its immediately superior directory. Client requests are sent to the master at the root of the tree. Each request percolates down the tree according to the directory names in the file specification. It eventually stops at the server responsible for the innermost directory. That agent fulfills the request. When a directory changes (e.g. a file is created or removed), the new directory content is checkpointed and the resulting UCR is passed upward in the agent hierarchy (toward the root). Whenever an agent receives a new UCR for a subdirectory's state, it updates and checkpoints its own state (i.e. own directory content) and passes the resulting UCR upward

in the hierarchy.

This extension promises improved efficiency due to increased parallelism. Instead of one entity manipulating a very large data structure, there are now a multitude of entities each maintaining a small portion of the overall data structure. Overhead for coordination of this parallel activity is also distributed among the entities.

Highlights of this directory scheme and its implementation are:

- Slaves and the master agent are instances of the same program.
- Slaves are only started for portions of the file hierarchy which are "active"; i.e. only for subtrees in which files are being created, accessed, or removed.
- Checkpointing still underlies storage operations.
- UCR concept minimizes the amount of information actually stored.
- When a directory changes, an "update" percolates up the tree. Updates are only performed by nodes on the shortest path to the root from the directory node at which the initial change occurred.
- Directories are automatically created as files within them are created. There is no explicit request to create a directory.
- A file and a subdirectory with the same name can co-exist within the same directory.

This further enhanced file system kernel has been implemented in FCP and tested under Logix. It again consists of an interface module and a main module. The program is given in Appendix D. The program also incorporates several extensions for improved performance.

6.3.6.2. Other Extensions

The concept of unique, compact representation could be used to reduce the cost of repeatedly storing file names. This extension was not pursued because it does not offer significant gain. Names of files are typically "smaller" than their contents. More importantly, file name lookup – an operation which occurs repeatedly – requires the expanded form of file names. The kernel would have to expand each name UCR checked during the lookup, or keep the expanded form in its data state. The result is more maintenance with little improvement in storage costs. If file names are hierarchical, the previous scheme demonstrates one successful approach in which complete names are not stored.

It is possible to store the history of each file's content, not just its current content. This can be done without excessive overhead using the UCR concept. A file, then, is not represented by *(FName,FContent)* but by *(FName,[CurrentFContent/PreviousFileHistory])*. That is, a file becomes a <file name, file history> pair, where the history is a list of terms which have been the file's content, the head of the list being the current content. Such a strategy is followed in some optical disk file systems [Garf85]. However, the extension is of questionable value. It is already possible to retrieve previous versions of a file through the file system history, and the need for a more rapid mechanism has not become apparent. Also, though conceptually straightforward, the modification complicates data structures and clauses of the file server.

Other possible extensions include

- dropping the requirement that file names be ground;
- instead of failing requests with non-ground file names, having them suspend until the name is fully instantiated;
- allowing functional terms of more than one argument as file names;
- having the third argument of the *affirm* request be a stream, where the current contents of the file are checkpointed each time a new element of the stream is added;
- making the contents of files streams rather than arbitrary terms;
- incorporating the functionality of *affirm* and *access* requests into a single request which will access the specified file if it exists, and otherwise create it.

The implications and properties of these alternatives have not been explored.

6.3.7. Summary

This section has presented enhanced file system designs. The enhancements rectify the problems given in Section 6.2.1 with the file system kernel and its naive implementation. Necessarily, the semantics of requests are altered from those in Sections 6.1 and 6.2; for example, extra arguments are added to some, and *access* cannot be used to extend a file. The resultant file systems retain many earlier, desirable characteristics such as: a file system kernel and file system servers cooperating to provide services, data transfer by unification, and checkpointing to a secondary storage device process to achieve nonvola-

tility. A number of noteworthy concepts result, including lazy term expansion. The unique, compact representation concept is crucial to the enhanced file system kernels; the utility of the concept is made obvious.

6.4. Other Work

The idea of providing nonvolatile storage by checkpointing is not unknown. In the Eden system [Blac83], files are active entities rather than data structures. A checkpoint primitive is the only operating system primitive by which files access stable storage.

"Logical file I/O" is being introduced to some (sequential) Prolog implementations. For example, Waterloo Unix Prolog ("WUP") [McCl88] supports file I/O without side-effects. Files are abstract streams of objects. They are manipulated in the style of lazily-evaluated lists. Predicates to perform file operations can be backtracked over.

As mentioned in Chapter 4, file systems have been developed for functional programming languages and architectures. Those schemes can exhibit similar properties, but are invariably less powerful due to the lack of the logical variable and necessity of tagged splits and merges.

The approach here has parallels with work on recoverable and persistent virtual memory [That86]. The latter uses checkpointing and garbage collection to provide a uniform memory abstraction. The distinction between transient and persistent objects in memory is eliminated; they are treated consistently and with no change in representation. Many parallels can be drawn between these persistent objects and files supported by CP objects. Implementation techniques for persistent objects may be applicable to logic-based secondary storage. As well, a file system kernel could be realized as a process (such as *fs* above) which is a persistent object. The local argument of the process, the file system state, would be stored in the persistent memory.

An interesting, related scheme for file storage [Clea84] involves turtle annotations for CP [Shap83b]. The turtle notation is extended to allow absolute locations: designations that a process must be executed on a specific processor. Using this idea, files are regarded as active processes created on demand. They modify their states in response to "read" and "write" messages. A "close" message, however, is processed as follows:

file([close/ReqStrm], FileContent) :–
file(ReqStrm?, FileContent)@disk.

That is, the *close* simply directs the process to the processor designated as *disk*. This processor stores[23] a nonvolatile representation of any (file) process which migrates to it for later retrieval and activation. This idea is promising, though many details have yet to be investigated.

The SIMPOS file system [HaYo84] provides permanent storage for the PSI prototype inference machine [UYYT83, YYTN83]. However, it is very similar to a conventional file system. A file consists of records, which are the units of I/O. File operations are achieved by imperative commands with side-effects. Conversion is often necessary between internal (within the machine) and external (on the storage medium) forms of information. No use is made of a logic-programming language in specification of the file system. Thus, though the file system is targeted for novel hardware, it displays few innovative characteristics.

6.5. Discussion, Comparison, and Conclusions

The core of a file system can be a perpetual CP process which maintains a database of <file name, file content> associations as part of its internal state. Portions of this local state, individual files, may be shared between the process and client processes. Transfer of information is through instantiation. The unique property of files – as opposed to terms in the local state of an ordinary perpetual process – is that they are nonvolatile after a certain characteristic point in the interaction between file system and client. A storage medium device process provides the means by which a file is made nonvolatile. This unusual and uncomplicated view of a file system is encouraged by the expressiveness and versatility of CP – and similar languages – and the characteristics of the machine model. In this chapter, emphasis has been on keeping this simple concept of storage, yet providing nonvolatility, adequate functionality, and acceptable performance.

The file system design advocated in this chapter combines a file system kernel and file servers. The kernel is responsible for basic file services, such as creation, retrieval, and elimination. File servers build on these services. Several alternate designs for the file system kernel have been presented. Examples of possible file servers were also given. All are user-level programs. The file systems described

23. This assumes that there is already some underlying structure to the information on the disk.

minimize problems due to side-effects, data conversion, and destructive assignment. Term-based, declarative I/O is provided.

The first file system kernel is simple, but sufficient. It maintains the entire file system data structure as a (CP) process's data state. The model's specification, with little modification, forms a concise, working implementation. However, it suffers from a number of problems. Most were a consequence of the implementation method used. The overall design is still clean, elegant, and powerful. The second kernel design avoided many of these problems. This was done at a cost of greater complexity. The enhanced kernel maintains an entire file system history. The file abstractions provided by the two kernels are very similar. Interesting aspects of both designs include: file names and contents as arbitrary terms (though names must be ground); unification-based data transfer, where active participation by the kernel (e.g. read and write requests) is unnecessary; storage of information onto nonvolatile media through checkpointing; and concise implementation programs. The second kernel also exhibits a new feature called lazy term expansion. It is facilitated by the unique, compact representation concept of the storage medium device process. The extended kernel is further augmented to support a hierarchical naming scheme. This last kernel is then implemented by a dynamic hierarchy of communicating agents where UCRs become even more important. Many of the interesting features of these successive designs are facilitated by the power of logic variables.

All file system kernel designs implicitly assume infinite storage capacity. Unbounded capacity is consistent with optical-disk technology. A result of this assumption is that no kernel uses destructive assignment, yet none requires garbage collection. Thus an entire file system history can be (conceptually) retained.

Checkpointing was used because it is a clean, yet powerful operation. A solution to the problems with checkpointing in the first kernel implementation (discussed in Section 6.2.1) was afforded by the introduction of a user indication to fix (freeze) a file's content. The operation can then be performed automatically and on a finer scale. However, it can be wasteful of storage space, especially if duplicate information is recorded. The unique, compact representation (UCR) concept was very important in this respect.

The UCR concept makes the enhanced file system kernel design practical. It reduces the overhead

of storing the file system history. Also, it allows a file to appear in multiple versions of the file system content economically. UCRs are also important for examination of stored information, facilitating lazy term expansion.

For a parallel logic-inference machine to be efficient access to shared terms must be efficient, the amount of information stored to accommodate large shared data structures must be minimized, and shared variable bindings must be disseminated rapidly. The efficiency of accessing files stored by the file system kernel is dependent on precisely these considerations. A similar case may be made for file system servers. Consequently, the degree to which these issues are resolved in a logic-based computer has direct bearing on the efficiency of the file system.

An interesting aspect of this model is that the file system becomes part of the interface between the environments of the machine's deductive mechanism (in which a client's variables exist) and the physical storage medium (on which the variables do not exist). This gives a further purpose to the file system within the computer system.

Given the expressiveness of concurrent logic-programming languages, a more ordinary file system design could have been provided. However, the opportunity was taken to develop an unconventional approach to file storage, which made better use of the remarkable properties of concurrent logic-programming languages and the machine model.

The enhanced file system kernel has many parallels with designs of optical-disk file systems for conventional computer systems [Garf85, Vitt85]. These similarities exist because the write-once property of these media is analogous to the single-assignment character of logic variables. For example, optical-disk file systems usually support access of the history of stored information. Their designs also usually assume infinite storage capacity.

The paradigm of persistent objects [That86] treats information consistently, irrespective of whether it is in main memory (and active with respect to a computation) or recorded on secondary storage media. The concept of file storage presented here is in keeping with this notion. A file is maintained as persistent information. That is, the file, or a representation of it, is maintained as part of the data state of a perpetual process. There is little distinction between accessing a file and accessing the information maintained by a "normal" CP object.

Much of this work has already been described in earlier papers, both in the context of CP [Kusa85b, Kusa87] and PARLOG [FoKu86].

6.5.1. Further Work

A number of avenues for further work have been identified:

- It is possible to drop the requirement that file names be ground. For example, file names could be required to be ground on *affirm* requests only. On *access* and *revoke*, they could be wholly or partially instantiated. As mentioned earlier, the properties of the resulting file system model have not as yet been explored.
- The file system model implicitly assumes that infinite storage resources are available. However, even though it may be very large, some finite limit must exist for storage capacity. A secondary storage device process can have finite resources, as outlined in Section 4.1.7.2. A client – in this case, a file system kernel – need only indicate when information becomes "old". For the first file system kernel, the situation is obvious: the previously checkpointed file system data structure becomes "old" when a new one is checkpointed. The case for the extended kernel is more complex. Other schemes for bounded capacity involving migration or transfer of "current" information to fresh media are also plausible. In any case, modification of the file system models and implementations to accommodate bounded resources warrants attention.
- The multi-agent file system kernel discussed in Section 6.3.5.1 has known shortcomings. For example, a directory continues to consume a portion of a file system content even after it has been deleted. Also, the search for matching file names on an *inventory* request is not recursive. Finally, though a file system history is maintained, no ability to access previous versions of a file has been incorporated. Rectifying such problems is a possibility for future work.
- Only rudimentary or unsophisticated file servers have been presented. As outlined before, more elaborate ones – possibly driven by actual application requirements – are worthy of effort.

6.5.2. Final Remarks

This chapter has further developed the idea of term-based, declarative I/O, and demonstrated its viability. The innovative approach to secondary storage begun in earlier chapters was continued. Two

interesting file system designs and implementations were discussed. They provided a test-bed for the secondary storage medium implementation described in the previous chapter. Problems typical of I/O operations in other logic-programming environments were minimized. Inefficient representational conversion was avoided. There was no need to introduce new constructs to CP, or make use of unique properties of the language. The designs and implementations were based on first-order logic, without recourse to side-effects or destructive assignment. The power of concurrent logic-programming languages was utilized. Logic variables and unification played a key role. The principle of executable specification was again shown.

7. Summary and Conclusions

This thesis has investigated how advantage can be taken of the features of logic-programming languages in the design and implementation of logic-based computer systems. The investigation centered on three main ideas: executable specification, declarative I/O, and implementation through transformation and meta-interpretation. Concurrent Prolog (CP) was the primary language of the thesis. It served as specification and machine language. Because neither real CP machines nor practical CP machine emulators yet exist, prototype implementations were carried out in a restricted dialect, Flat Concurrent Prolog (FCP).

This chapter summarizes and concludes the thesis. A review of the work, with highlights of each chapter, is presented in Section 7.1. The thesis objectives are restated and conclusions drawn. Section 7.2 considers the future: both application of results to other areas, and promising topics for further investigation.

Related work of others has been covered in relevant context within each chapter. It will not be repeated here.

7.1. Review with Conclusions

Detailed objectives for the thesis were given in Section 1.3. They are summarized as follows:

- determination of hardware, software, and overall system models for a logic-based computer, and examination of the model's noteworthy aspects;
- derivation of executable specifications for the above models;
- a method for initiating a logic-based computer system;
- development of models of peripheral devices whose actions and interactions can be described without side-effects or destructive assignment, and whose units of transfer are terms (i.e., devices supporting declarative, term-based I/O);
- a methodology for realizing peripheral device implementations from their model specifications;
- construction of a file system model that also supports declarative I/O, and its implementation;
- demonstration and exploitation of the features of logic-programming languages, such as unification and the capability for executable specification.

This section reviews how these objectives were satisfied in the previous chapters, and draws conclusions based on that work. Important, innovative concepts which arose are emphasized.

Initially, a flexible hardware architecture and operating system design were presented. These were assumed for the remainder of the thesis. A highlight of the architecture was the device process and device processor concepts. These concepts later formed the basis for declarative interactions with peripheral devices. The operating system design followed principles of multi-process structuring, server-based operations, and object-based communications. Many of the features of the design were made possible by properties of parallel logic-programming languages.

Models for a logic-based computer system were presented in Chapter 3. It was demonstrated that CP programs can concisely and elegantly specify a computer system, its operating system, and the operation of its peripheral devices. The computer system is represented as a logic-programming goal *computer_system*. Clauses used to resolve the goal – and ensuing subgoals – progressively refine the hardware, operating system, and computer system models. Interfaces between peripheral devices and operating system are also captured. The accumulation of all clauses describing the logic operating system constitute its implementation. Computer systems with vastly differing characteristics can be concisely specified in this manner. Two possibilities were explored in detail. They differ in style of communication between operating system and peripheral devices, representation of errors, and ability to restart operation. Two fundamentally different types of systems and peripheral devices were identified: perpetual and restartable. The former type has no provision for reinitialization once initiated; its operation must be seen to span hardware errors and inactive periods. Restartable systems and devices, on the other hand, can be reinitiated. Peripheral devices may be conceptualized as either type, though some are better suited to one type. A nondeterministic oracle process is useful in specifying as restartable a device with persistent content or state (e.g. nonvolatile memory). Care must be taken in mixing perpetual and restartable devices in a system. A problem that may otherwise arise was exemplified. Thus, models of computer systems were developed which can guide the genesis of actual logic-based computers. These models exercised and accentuated features of logic-programming languages.

It was shown that the clauses specifying a logic-based computer system also describe the initialization procedure for the system. This capability is due to the dual operational/declarative nature of logic-

programming languages. The procedure is keyed upon coordinated access to predetermined, shared logic variables. A need for primary and secondary bootstraps can be accommodated without violating the correspondence between model refinement and initiation procedure.

Two techniques for developing models of peripheral devices which support term-based, declarative I/O were presented in Chapter 4. The need for, and utility of, this form of I/O was justified. For continuity, a restartable, nonvolatile secondary storage medium (a magnetic disk) was chosen as a representative example. The first model development technique covered views I/O as interaction between independent logic (inference) systems. The second regards a peripheral device as a computation of a function. Both techniques allow the state or content of a device to be explicitly represented.

With the independent logic system technique, a computer is composed of two or more distinct computational nodes. Each node is a logic-based system in its own right, possessing a knowledge base, inference mechanism, and communication mechanism. A secondary storage device was modelled as such a node. Its contents constitute a knowledge base defining a single predicate. The device can accept assertions from other nodes of new facts (storage of information) and queries for existing facts (retrieval). Unique identifiers (IDs) for terms were introduced to eliminate costly pattern-based searches in response to queries. For practical considerations, sharing of logic variables between storage device and other nodes must be prevented. Two schemes were considered: requiring all information (terms) to be made ground (i.e. contain no variables) before storage, and renaming variables on assertion or query. The schemes yielded comparable device models, though the latter scheme allows stored information to be augmented. Both naturally led to a concept of "environment" relating to the scope of logic variables. Specifications for storage devices following each of the two schemes were given and scrutinized. The storage models developed using the independent logic system technique have no provision for retracting assertions (deleting or replacing stored information). This greatly simplifies the models. It also implicitly requires unbounded storage capacity. It was shown that the storage models can be revised to support finite resources, reclamation of space, and modification of stored information. The revision follows the event calculus [KoSe86] and introduces time stamps. The storage device records histories (of facts), rather than just simple facts. A specification of the resulting storage device was given and examined. As was also discussed, the independent logic system technique is readily applicable to other peripheral devices.

The second technique for developing device models views declarative I/O as the computation of a function (restricted relation) between identifiers and terms. The memory content of a secondary storage device encodes previously computed members of the function. Two types of interaction are supported. To access stored information, an identifier is given, and the corresponding term (such that the identifier-term pair is a member of the function) is sought. To record information, a term is given, and a corresponding identifier determined. The new identifier and stored term are added, as another member of the function, to the device content. To satisfy the conditions of a function, stored information must be made ground. For the same reason, the memory is prevented from becoming read-only after some number of archiving operations by assuming infinite capacity. A specification for a storage device following this model was given, and its characteristics examined. The assumption of unbounded resources can be dropped if a relation between identifiers and terms is computed, rather than a function. This variation of the function-computation technique was illustrated by constructing a model of a finite-capacity storage medium. The model has many commonalities with its infinite-capacity counterpart. The versatility of the function-computation technique was also demonstrated by deriving models of video output and keyboard input devices.

The disk storage models produced by the two techniques have many similarities, though there are subtle, but fundamental, differences. With both, the interface to a device is achieved by declarative mechanisms. The processing of storage and retrieval operations is describable by logic, and is thus easy to specify, modify, and reason about. Identifiers figure prominently in the two techniques. For disks, an identifier is a short-hand representation of one, and only one, recorded term. This concept – unique, compact representation (UCR) – is an innovative contribution of the thesis. The concept is also relevant to I/O involving other devices. It proved invaluable in subsequent discussions. An attempted retrieval using a nonexistent identifier illustrates one difference between similar models produced by the two techniques. Under the first technique, failure results; with the second, the query suspends, possibly indefinitely. Other examples of differences were also given.

Even though the device specifications in Chapters 3 and 4 are executable, they are not practical implementations directly. Chapter 5 thus pursued a methodology for realizing an implementation from a specification program. The target language was FCP. A secondary storage device was again the representative example. The first stage of the methodology consists of a sequence of enhancements and

equivalence-preserving transformations. The enhancements augment the client interface, but do not change the basic device model. After modification, the specification program accesses stored information (the device content) in a restricted, localized manner. This allows the device content and its manipulation operations to be moved to a separate perpetual process. The specification in CP is then translated to FCP. The process manipulating the device content can be promoted to the "meta-level" (the same execution level as a FCP interpreter executing the specification program). The change in the FCP execution mechanism is describable by enhancements to a FCP meta-circular interpreter. To reflect this enhanced functionality, the existing FCP emulator is augmented with two system predicates. These predicates have well-defined semantics and can be understood declaratively. Their units of transfer are terms. The implementation of the storage device consists of the enhanced FCP execution mechanism and the final FCP specification program. A demonstrable, working prototype was realized in this way as part of the thesis research. It was argued that a real, production device based on this prototype is feasible. Though the methodology was applied to only one of the storage device models in Chapter 4, it is amenable to other models (e.g. one based on function computation), to other devices, and alternate languages. However, complications can arise for some models, and its feasibility for devices with nondeterministic specification programs is uncertain. Chapter 4 thus describes a prototype implementation of a device process for a nontrivial computer peripheral. The implementation methodology is novel in its application of meta-interpretation and source-to-source transformation, two important concepts in logic programming.

Chapters 4 and 5 demonstrated that declarative, term-based I/O is a practical concept. Models of nontrivial devices supporting it were developed and implemented. Undesirable characteristics more typical of I/O operations – reliance on side-effects, use of destructive assignment, and inefficient conversion of representation – were avoided. Features of logic-programming languages played an important role. Machine-level data transfer was achieved through unification. The (dataflow) synchronization mechanism of restricted unification replaced interrupts. Since a magnetic disk was the representative example in the two chapters, the thesis has in effect presented an innovative approach to nonvolatile secondary storage in a logic-programming environment.

In Chapter 6 another nontrivial computer system component, a file system, was developed. The objective was to further demonstrate the principles of declarative I/O and executable specification. The

basic file system design consists of a file system kernel and file system servers. The kernel provides all basic file system capabilities; for example, file creation, deletion, and access. File servers enhance these basic capabilities. A working implementation of a logic-based file system was the result of several phases of development. The discussion concentrated on file system kernels, though examples of servers were also presented. The initial file system kernel was simply an unspecialized perpetual process which maintains a database of <file name, file content> associations. A file is manipulated, as any term, through unification. An executable specification of the kernel was given, though as an implementation program it does not provide nonvolatility of storage. This shortcoming was corrected by introducing a checkpoint operation and utilizing the secondary storage facilities developed in Chapter 5. The use of checkpointing did not alter the basic model, and an initial implementation was thus achieved. The next stage of development was to make the file system kernel more practical and efficient. The kernel was extended to retain a history of the file system content, rather than just an instantaneous version. As stable storage was still provided through checkpointing, this new kernel was potentially wasteful of storage resources. Fortunately, the UCR concept formulated earlier curtails the problem. For example, the UCR of a file content term can be stored in successive versions of the file system instead of the actual term. This extended model, and a working implementation, were described at length. Further enhancements were also discussed. One, to support hierarchical directories, constituted the last stage of file system development. Hierarchical directories were realized by making the file system kernel a multi-process tree of "agents". Actual use of the file system implementations confirmed their practicality.

Interesting features of the logic-based file systems developed include the following. There are no explicit read or write operations because files are accessed and manipulated through unification. The UCR concept curbs potential inefficiencies. It also facilitates lazy term expansion, where a term (typically the content of a file) is normally encoded as a UCR (or part of a UCR) and is "expanded" only when desired or required. The secondary storage models discussed in Chapter 4 introduced a concept of environments over which variables are known. The storage device is a separate environment, and a file system kernel is seen as part of the interface to it.

Throughout the file system presentation emphasis was on keeping a single, simple paradigm of storage, yet providing nonvolatility and efficiency. The model was that a file is a part of a data structure maintained by a perpetual CP process. Portions of the data structure may be shared by other processes.

The unique property of files – as opposed to shared terms in other data states – is that they are nonvolatile after a certain characteristic point in the interaction between perpetual process and client. A secondary storage device process provides the means by which a file is made nonvolatile. This simple view of file storage factors away many of the details which usually complicate file systems, but, based on this work, may be unnecessary. It eliminates a common dichotomy in logic-programming environments, where dissimilar models exist for accessing information stored on disk or stored within executing programs. Phrased in more standard operating systems' terminology, the model views entities (terms) in file storage or in main memory as equivalent except for speed of access and volatility.

Infinite storage capacity was assumed for the representative secondary storage device used throughout much of the thesis. This assumption is reasonable given the large – though finite – capacities of write-once, optical disks. This write-once character is analogous to the single-assignment property of logic variables. The operation of such devices is therefore well-matched by logic programs. Conventional file systems for write-once optical disks have been based on the assumption of infinite capacity [Garf85, Vitt85]. Such file systems have much in common with the logic-based file systems presented in Chapter 6. The advantage of the write-once property in the conventional systems is that if storage costs are cheap enough, capacity large enough, and retrieval fast enough, there is no reason to delete information. That property has another benefit in logic-programming environments: it prevents accidental, deliberate, or covert use of destructive assignment.

This thesis has illustrated the power and utility of logic-programming languages, particularly committed-choice forms supporting parallel computation. Throughout, their applicable features were highlighted and exploited. Benefits of this exploitation were stressed. For example, Chapters 3, 4, and 6 demonstrated the capability for executable specification; specification programs were used directly to provide implementations. Chapter 5 showed the added benefit of applying transformations. The specification programs lucidly expressed parallel computation, process synchronization and coordination, hierarchical constitution, data construction and decomposition, and data transfer and manipulation. One important aspect of the work is that a single, uniform formalism could be used throughout.

CP has served well as a machine, specification, and implementation language. Particularly useful features of the language were "light-weight" processes, dynamic process creation and destruction,

stream-based interprocess communication, logic variables, flexible yet controlled sharing of data structures, dataflow synchronization, AND-parallelism, and (don't-care) nondeterminism. Many of these features are shared by other languages – PARLOG is one example – so the results of this thesis are not specific to Concurrent Prolog.

In summary, this thesis has shown that innovative and advantageous approaches to software, hardware, and computer system design and implementation are feasible in a logic-programming context. Many examples were presented. The thesis has demonstrated how a logic-based computer system can be both specified and implemented using a logic-programming language, how interactions with peripheral devices can be described declaratively, and how such interaction can be realized for a nontrivial device and significant computer service. Innovative concepts resulting from this work include device processes and device processors; restartable and perpetual devices and systems; peripheral devices modelled on function computation or as independent logic systems; unique, compact representations of terms; lazy term expansion; file systems through perpetual processes; and unification-based file operations.

Various subsets of the material in this thesis have already appeared. This includes the specification and initialization of computer systems [Kusa86], a secondary storage model using one of the techniques in Chapter 4 , [FoKu86] emulator modifications necessary for prototype storage device implementation [FoKu86], and the logic-based file system [Kusa85b, Kusa87]. However, a great deal of additional, fresh material was also covered. This work was a complete, unified treatment of all these results.

7.2. Application and Extension of Results

Concepts developed in this thesis can be, and have been, applied in other areas. Some examples are given. The application areas range from closely related (to that of the thesis), to distantly related.

- The PARLOG Programming System (PPS) has already made use of the declarative secondary storage device model of Section 4.1.3 and the implementation techniques in Section 5.4 [Fost87c]. These were used for accessing disk storage in the implementation of PPS.
- The term-based, declarative I/O facilities can be used in the development of user environments for prototype logic-based computers. Such user environments do not always incorporate ideas of declarative I/O. SIMPOS [HaYo83] is one example. Sequential Prolog implementations are moving

in the direction of providing more declarative (e.g. backtrackable) I/O facilities [McCl88]. Works such as this thesis also encourage that trend.

- Composition of a logic-based computer system from smaller independent logic-based systems (Section 4.1) can influence development of user environments for such computers. In particular, it may be advantageous to view the user as an independent logic-based system. This would affect the manner in which the user accomplishes tasks and communicates with system components.
- An obvious application area is the production of parallel logic-inference machines and their operating systems. All the results of this thesis are relevant to, and useful for, such work.
- As discussed in Section 6.4, the file abstraction presented here has strong parallels with ideas of "persistent objects" [That86]. Thus, the models and implementation techniques employed may be useful in the development of systems according to the persistent-object paradigm.
- Functional operating systems, purely functional abstract machines [Jone84b], and experimental functionally-programmed multiprocessors [Youn85] bear resemblance to their counterparts in logic programming. Thus, concepts presented here, such as peripheral devices performing function computation and the implementation methodology in Chapter 5, may be relevant to the functional context. The power of the logic variable was crucial to many concepts, however. Because logic variables are absent from the functional-programming paradigm, applicability of some results may be hampered.
- Ideas from this thesis are also applicable to conventional, von Neumann architectures and their operating systems. For example, the elegance of the operating system design based on multiprocess-structuring, message passing, and servers lends credence to those concepts in the von Neumann context. Conventional systems could benefit from concepts such as declarative I/O (to facilitate verification, for example). By using explicit, functional read and write operations for data transfer (instead of unification), the logic-based file system models could be adapted.

More applications are mentioned in the remainder of this discussion.

A great deal of further, interesting work is suggested and motivated by this research. Some involves areas already explored, some delves into new areas. A few possibilities for future investigation are given here. Many others have already been mentioned in relevant chapters.

Initially, a prospective hardware architecture and operating system design were outlined. These can be developed to a greater level of detail. The simplicity of the O/S design, and the fact that it can so easily be captured with a declarative specification, suggest that it is composed of "fundamental" constructs. It may be that all "good" operating systems must be consistent with these constructs. This would require investigation to substantiate.

The operating system design shows promise for providing computer security. A typical model for a "secure computer system" involves isolation of processes, with communication among them restricted to specific channels [Giff82, Lamp73, Rush81]. CP supports isolation between computational entities because variables are local to a process and cannot be unknowingly shared by another process. Some work has already been done in this area [MBTL87]. The well-behaved nature of device processes also makes the idea of a filtering device [Denn79] for security attractive.

In the storage medium model of Section 4.1, the device process only dealt with assertions and queries of facts defining the single predicate, *disk*. Extension of this capability, to full meta-level database maintenance [BoKo82, BoWe85], could be explored. This would integrate with research in relational and deductive databases [Mink86].

The specifications of computer system components did not involve side-effects or destructive assignment. Verification of the specifications is thus an obvious future pursuit. Of particular interest is showing the correctness of a logic-based operating system.

Infinite capacity for secondary storage was assumed in Chapters 4, 5, and 6. Actual storage resources are necessarily finite, however. Finite storage was addressed in most discussions, yet significant opportunities for further examination still exist. Section 4.1.7, for example, described how a bounded-capacity storage device could be modelled using the independent logic system technique and ground-term storage approach. However, a corresponding treatment was not performed for the term-copy approach. A prototype implementation of a finite storage device could be pursued. The effect of eliminating the infinite-capacity assumption on the implementation methodology of Chapter 5 would be most interesting. Finally, a the logic-based file system could be modified for finite storage.

Another possibility for future investigation is invention and exploration of new peripheral devices having interesting characteristics or useful functionality. Models could be formulated, specified, and

implemented according to the procedures illustrated in this thesis. A real hardware realization could even be pursued. An example device might be a video display and modest-capacity disk combination, where the screen acts as a "window" on the last information received (i.e. a terminal with significant off-screen memory).

The implementation of the storage medium device yielded possibilities for further work. An implementation in a more distributed environment (e.g. a FCP emulator on a multiprocessor) is one. A separate processor could be dedicated to the storage medium device process. This would expose any hidden single-processor dependencies in the current implementation and emulator. A distributed implementation should also be more efficient.

A last subject for follow-up investigation is the file system of Chapter 6. A small, but comprehensive, set of file system servers could be developed.

7.3. Final Remarks

This thesis can be represented by the logic-programming goal

research(BackgroundResults?, InterestingResults)

where *BackgroundResults* is provided a binding by researchers such as Gregory, Shapiro, Takeuchi, Ueda, Foster, and many others. The goal (eventually) succeeds, generating a binding for the stream *InterestingResults*. The stream has many elements, but in true logic-programming fashion, its tail is uninstantiated. This tail is the variable *FurtherWork*. It is now up to another goal (whose solution may have already commenced) to further instantiate *FurtherWork*.

References

Abad86.
M. Abadi, "Temporal Theorem Proving," Ph.D. Thesis Dissertation, Department of Computer Science, Stanford University, December 1986.

AhHU83.
A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

Aiso82.
H. Aiso, "Fifth Generation Computer Architecture," in *Fifth Generation Computer Systems*, ed. T. Moto-oka, pp. 121-127, North-Holland, Tokyo, Japan, 1982.

ArIa83.
Arvind and R. A. Iannucci, "Two Fundamental Issues in Multiprocessing: the Data Flow Solution," *Proceedings of the 10th International Symposium on Computer Architecture*, Stockholm, Sweden, June 14–17, 1983.

BCMD87.
W. R. Bush, G. Cheng, P. C. McGeer, and A. M. Despain, "Experience with Prolog as a Hardware Specification Language," *Proceedings, 1987 Symposium on Logic Programming*, pp. 490–498, IEEE, San Francisco, August 31 – September 4, 1987.

Bic84 .
L. Bic, "Execution of Logic Programs on a Dataflow Architecture," *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 290-296, IEEE, Ann Arbor, Michigan, June 5-7, 1984.

Blac83.
A. P. Black, "An Asymmetric Stream Communication System," *Proceedings of the Ninth Symposium on Operating Systems Principles*, pp. 4-10, ACM, Bretton Woods, New Hampshire, October 10-13, 1983.

BLMO86.
R. Butler, E. Lusk, W. McCune, and R. Overbeek, "Parallel Logic Programming for Numeric Applications," *Proceedings, 3rd International Conference on Logic Programming*, London, July 14-18, 1986.

Bloc84.
C. Bloch, "Source-to-Source Transformations of Logic Programs," CS84-22, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, December 1984.

BoKo82.
K. A. Bowen and R. A. Kowalski, "Amalgamating Language and Metalanguage in Logic Programming," in *Logic Programming*, ed. K. L. Clark and S.-Å. Tärnlund, pp. 153-172, Academic Press, London, England, 1982.

Bowe82.
Bowen, "Concurrent Execution of Logic," *Proceedings of the First International Logic Programming Conference*, pp. 26-30, Marseille, France, September 14-17, 1982.

BoWe85.
K. A. Bowen and T. Weinberg, "A Meta-Level Extension of Prolog," *1985 International Symposium on Logic Programming*, pp. 48-53, IEEE, Boston, Massachusetts, July 15-18, 1985.

Broo87.
F. P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, vol. 20, no. 4, pp. 10-20, April 1987.

Cher79.
D. R. Cheriton, "Multi-Process Structuring and the Thoth Operating System," Tech. Rep. 79-5, Computer Science Department, University of British Columbia, Vancouver, B.C., Canada, March

1979.

Chik83.

T. Chikayama, "ESP - Extended Self-contained Prolog - as a Preliminary Kernel Language of Fifth Generation Computers," *New Generation Computing*, vol. 1, no. 1, pp. 11-24, Tokyo, 1983.

ChZw83.

D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth Symposium on Operating Systems Principles*, pp. 129-139, ACM, Bretton Woods, New Hampshire, October 10-13, 1983.

ChZw84.

D. R. Cheriton and W. Zwaenepoel, "One-to-Many Interprocess Communication in the V-System," *Proceedings of the 4th International Conference on Distributed Systems*, ACM, February 1984.

Clea84.

J. G. Cleary, April 1984. Personal communication.

ClGr81.

K. L. Clark and S. Gregory, "A Relational Language for Parallel Programming," *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architectures*, pp. 171-178, ACM, October 1981.

ClGr83.

K. L. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," Research Rep. DOC 83/5, Department of Computing, Imperial College, London, May 1983.

ClGr84a.

K. L. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," Research Rep. DOC 84/4, Department of Computing, Imperial College, London, April 1984.

ClGr84b.

K. L. Clark and S. Gregory, "Notes on Systems Programming in PARLOG," Research Rep. DOC 84/15, Department of Computing, Imperial College, London, July 1984.

ClGr86.

K. L. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 1, January 1986.

ClMe81.

W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.

ClMG82.

K. L. Clark, F. G. McCabe, and S. Gregory, "IC-PROLOG Language Features," in *Logic Programming*, ed. K. L. Clark and S.-Å. Tärnlund, pp. 253-266, Academic Press, London, England, 1982.

CoKi81.

J. S. Conery and D. F. Kibler, "Parallel Interpretation of Logic Programs," *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pp. 163-170, ACM, Portsmouth, New Hampshire, October 18-22, 1981.

Colm86.

A. Colmerauer, "Theoretical Model of Prolog II," in *Logic Programming and Its Applications*, ed. M. van Caneghem and D. H. D. Warren, pp. 3-31, Norwood, New Jersey, 1986.

CoSh86.

M. Codish and E. Shapiro, "Compiling OR-parallelism into AND-parallelism," *Proceedings, 3rd International Conference on Logic Programming*, pp. 283-297, London, July 14-18, 1986. Also available as TR CS85-18, Department of Applied Mathematics, Weizmann Institute of Science, December 1985.

DaBu76.

J. Darlington and R. M. Burstall, "A System which Automatically Improves Programs," *Acta Informatica*, vol. 6, 1976.

Davi82.

R. E. Davis, "Runnable Specification as a Design Tool," in *Logic Programming*, ed. K. L. Clark and S.-Å. Tärnlund, pp. 141–149, Academic Press, London, England, 1982.

Denn79.

D. E. Denning, "Secure Personal Computing in an Insecure Network," *C. ACM*, vol. 22, no. 8, pp. 476–482, August 1979.

Dijk76.

E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

EdSh84.

S. Edelman and E. Shapiro, "Quadtrees in Concurrent Prolog," CS84-19, Department of Applied Science, Weizmann Institute of Science, Rehovot, Israel, August 1984 (revised January 1985).

EiKM82.

N. Eisinger, S. Kasif, and J. Minker, "Logic Programming: A Parallel Approach," *Proceedings of the First International Logic Programming Conference*, pp. 71-77, Marseille, France, September 14-17, 1982.

EmGo84.

M. H. van Emden and R. G. Goebel, "Waterloo Unix Prolog User's Manual," Version 1.2, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, August 1984.

EmLu82.

M. H. van Emden and G. J. de Lucena, "Predicate Logic as a Programming Language for Parallel Programming," in *Logic Programming*, ed. K. L. Clark and S.-Å. Tärnlund, pp. 189-198, Academic Press, London, England, 1982.

ErRa84.

L.-H. Eriksson and M. Rayner, "Incorporating Mutable Arrays Into Logic Programming," *Proceedings of the Second International Logic Programming Conference*, pp. 101-114, Uppsala, Sweden, July 2-6, 1984.

FGRS86.

I. T. Foster, S. Gregory, G. A. Ringwood, and K. Satoh, "A Sequential Implementation of PARLOG," *Proceedings, 3rd International Conference on Logic Programming*, pp. 149-156, London, July 14-18, 1986.

Flei83.

B. D. Fleisch, "Operating Systems: A Perspective On Future Trends," *Operating System Review*, vol. 17, no. 2, April 1983.

FoKu86.

I. T. Foster and A. J. Kusalik, "A Logical Treatment of Secondary Storage," *Proceedings, Third Symposium on Logic Programming*, pp. 58-67, IEEE, Salt Lake City, Utah, September 21-25, 1986.

Fost87a.

I. Foster, "The PARLOG Programming System User Guide / Reference Manual," Version 0.4, Department of Computing, Imperial College, London, May 1987.

Fost87b.

I. T. Foster, "Logic Operating Systems: Design Issues," *Proceedings, Fourth International Conference on Logic Programming*, pp. 910–926, Melbourne, Australia, May 25–29, 1987.

Fost87c.

I. T. Foster, August 1987. Personal communication.

FuKM82.
K. Furukawa, N. Katsumi, and Y. Matsumoto, "Prolog Interpreter Based on Concurrent Programming," *Proceedings of the First International Logic Programming Conference*, pp. 38-44, Marseille, France, September 14-17, 1982.

Fuch83.
K. Fuchi, "The Direction the FGCS Project Will Take," *New Generation Computing*, vol. 1, no. 1, pp. 3-10, Tokyo, 1983.

FuTK83.
K. Furukawa, A. Takeuchi, and S. Kunifuji, "Mandala: A Concurrent Prolog Based Knowledge Programming Language / System," TR-029, ICOT, Tokyo, November 1983.

Garf85.
S. L. Garfinkel, "A File System for Write-Once Compact Disks," Research Report, MIT Media Lab, 1985.

Giff82.
D. K. Gifford, "Cryptographic Sealing for Information Secrecy and Authentication," *C. ACM*, vol. 25, no. 4, pp. 274–285, April 1982.

GoMe86.
J. A. Goguen and J. Meseguer, "EQLOG: Equality, Types, and Generic Modules for Logic Programming," in *Logic Programming, functions, relations, and equations*, ed. G. Lindstrom, pp. 295–363, Prentice-Hall, 1986.

GoTM84.
A. Goto, H. Tanaka, and T. Moto-oka, "Highly Parallel Inference Engine PIE - Goal Rewriting Model and Machine Architecture," *New Generation Computing*, vol. 2, no. 1, pp. 37-58, Springer-Verlag, 1984.

Greg85.
S. Gregory, 1985. Personal communication

HaYo83.
T. Hattori and T. Yokoi, "Basic Concepts of the SIM Operating System," *New Generation Computing*, vol. 1, no. 1, pp. 81-85, Tokyo, Japan, 1983.

HaYo84.
T. Hattori and T. Yokoi, "The Concepts and Facilities of SIMPOS File System," TR-059, ICOT, Tokyo, Japan, April 1984.

HeMa86.
R. Helm and K. Marriot, "Declarative Graphics," *Proceedings, 3rd International Conference on Logic Programming*, pp. 513-527, London, July 14-18, 1986.

Hend82.
P. Henderson, "Purely Functional Operating Systems," in *Functional Programming and Its Applications*, ed. P. Henderson and D. A. Turner, Cambridge University Press, 1982.

HeSh84.
L. Hellerstein and E. Y. Shapiro, "Implementing Parallel Algorithms in Concurrent Prolog: The MAXFLOW Experience," *1984 International Symposium on Logic Programming*, pp. 99-115, IEEE, Atlantic City, NJ, February 6-9, 1984.

HiCF84.
H. Hirakawa, T. Chikayama, and K. Furukawa, "Eager and Lazy Enumerations in Concurrent Prolog," *Proceedings of the Second International Logic Programming Conference*, pp. 89-100, Uppsala, Sweden, July 2-6, 1984.

Hira83.
H. Hirakawa, "Chart Parsing in Concurrent Prolog," TR-008, ICOT, Tokyo, Japan, May 1983.

HiSS86.

M. Hirsch, W. Silverman, and E. Shapiro, "Layers of Protection and Control in the Logix System," CS86-19, Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel, 1986.

Hoar78.

C. A. R. Hoare, "Communicating Sequential Processes," *CACM*, vol. 21, no. 8, pp. 666-677, August 1978.

Hogg82.

C. J. Hogger, "Concurrent Logic Programming," in *Logic Programming*, ed. K. L. Clark and S.-Å. Tärnlund, pp. 199-211, Academic Press, London, England, 1982.

HoSh86.

A. Houri and E. Shapiro, "A Sequential Abstract Machine for Flat Concurrent Prolog," CS86-20, Weizmann Institute of Science, Rehovot, Israel, 1986 .

JaLa87.

J. Jaffar and J.-L. Lassez, "Constraint Logic Programming," *Proeedings of the 14th Principles of Programming Languages Conference*, Munich, January 1987.

JaLM86.

J. Jaffar, J.-L. Lassez, and M. J. Maher, "Some Issues and Trends in the Semantics of Logic Programming," *Proceedings, 3rd International Conference on Logic Programming*, pp. 223-239, London, July 14-18, 1986.

Jone84a.

S. B. Jones, "A Range of Operating Systems Written in a Purely Functional Style," TR 16, Department of Computer Science, University of Stirling, Stirling, Scotland, September 1984.

Jone84b.

S. B. Jones, "Abstract Machine Support for Purely Functional Operating Systems," TR 15, Department of Computer Science, University of Stirling, Stirling, Scotland, September 1984.

Kahn82.

K. M. Kahn, "Intermission - Actors in Prolog," in *Logic Programming*, ed. K. L. Clark and S.-Å. Tärnlund, pp. 213-228, Academic Press, London, England, 1982.

KHKH87.

M. Kishimoto, A. Hosoi, K. Kumon, and A. Hattori, "An Evaluation of the FGHC Via Practical Application Programs," *Proceedings, 1987 Symposium on Logic Programming*, pp. 516–525, IEEE, San Francisco, August 31 – September 4, 1987.

KoSe86.

R. A. Kowalski and M. Sergot, "A Logic-based Calculus of Events," *New Generation Computing*, vol. 4, no. 1, pp. 67-95, Tokyo, 1986.

Kowa74.

R. A. Kowalski, "Predicate Logic as a Programming Language," *IFIP 74*, pp. 569-574, 1974.

Kowa79.

R. A. Kowalski, "Algorithm = Logic + Control," *C.ACM*, vol. 22, no. 7, pp. 424-436, July 1979.

Kowa82.

R. A. Kowalski, "Logic as a Computer Language," in *Logic Programming*, ed. K. L. Clark and S.-Å. Tärnlund, pp. 3-16, Academic Press, London, England, 1982.

Kowa85.

R. A. Kowalski, "Logic-based Open Systems," Research Report, Department of Computing, Imperial College, London, September 1985.

KTMB86.

K. Kahn, E. D. Tribble, M. S. Miller, and D. G. Bobrow, "Objects in Concurrent Logic Programming Languages," *Proceedings OOPSLA '86*, 1986.

Kurs86.

P. Kursawe, "How to Invent a Prolog Machine," *Proceedings, 3rd International Conference on Logic Programming*, pp. 134-148, London, July 14-18, 1986.

Kusa84a.

A. J. Kusalik, "Bounded-Wait Merge in Shapiro's Concurrent Prolog," *New Generation Computing*, vol. 2, no. 2, pp. 157-169, Springer-Verlag, Tokyo, Japan, 1984.

Kusa84b.

A. J. Kusalik, "Serialization of Process Reduction in Concurrent Prolog," *New Generation Computing*, vol. 2, no. 3, pp. 289-298, Springer-Verlag, Tokyo, Japan, 1984.

Kusa85a.

A. J. Kusalik, "Towards the Realization of a Logic Operating System," Ph.D. thesis proposal, University of British Columbia, Vancouver, B.C., February 1985.

Kusa85b.

A. J. Kusalik, "The File System of a Logic Operating System," Technical Report 84-21, Computer Science Department, University of British Columbia, Vancouver, B.C., Canada, November 1984 (Revised March 1985).

Kusa86.

A. J. Kusalik, "Specification and Initialization of a Logic Computer System," *New Generation Computing*, vol. 4, no. 2, pp. 189-209, Springer-Verlag, Tokyo, Japan, May 1986.

Kusa87.

A. J. Kusalik, "Secondary Storage in a Concurrent Logic Programming Environment," TR 87-5, Department of Computational Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada, 1987. To appear in *The Journal of Systems and Software* (accepted July, 1987).

Lamp73.

B. W. Lampson, "A Note on the Confinement Problem," *C. ACM*, vol. 16, no. 10, pp. 613–615, October 1973.

LeGo85.

R. K. S. Lee and R. Goebel, "Concurrent Prolog in a Multi-process Environment," *1985 International Symposium on Logic Programming*, pp. 100-109, IEEE, Boston, Massachusetts, July 15-18, 1985.

Levy84.

J. Levy, "A Unification Algorithm for Concurrent Prolog," *Proceedings of the Second International Logic Programming Conference*, pp. 331-341, Uppsala, Sweden, July 2-6, 1984.

Lock79.

T. W. Lockhart, "The Design of a Verifiable Operating System Kernel," Tech. Rep. 79-15, Computer Science Department, University of British Columbia, Vancouver, B.C., Canada, November 1979.

MaMW86.

Y. Malachi, Z. Manna, and R. Waldinger, "TABLOG: A New Approach to Programming Language," in *Logic Programming, functions, relations, and equations*, ed. G. Lindstrom, pp. 365–394, Prentice-Hall, 1986.

Marc84.

J. Marcure, "An Alvey Survey: Advanced Information Technology in the U.K.," *Future Generation Computer Systems*, vol. 1, no. 1, pp. 69-78, July 1984.

MBSD83.

M. Malcolm, B. Bonkowski, G. Stafford, and P. Didur, "The Waterloo Port Programming System," Technical Report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, January 1983.

MBTL87.

M. S. Miller, D. G. Bobrow, E. D. Tribble, and J. Levy, "Logical Secrets," *Proceedings, Fourth International Conference on Logic Programming*, pp. 704–728, Melbourne, Australia, May 25–29, 1987.

McCl88.

D. J. McClurkin, *WUP Version 3.0 User's Manual*, Department of Computer Science, University of Waterloo, January 4, 1988.

McCo83.

P. McCorduck, "Introduction to the Fifth Generation," *CACM*, vol. 26, no. 9, pp. 629-630, September 1983.

Mier84.

C. Mierowsky, "Design and Implementation of Flat Concurrent Prolog," CS84-21, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, December 1984.

Mink86.

J. Minker, ed., *Preprints of the Workshop on Foundations of Deductive Databases and Logic Programming*, University of Maryland, Washington, D.C., August 18–22, 1986.

MiTC85.

T. Miyazaki, A. Takeuchi, and T. Chikayama, "A Sequential Implementation of Concurrent Prolog Based on the Shallow Binding Scheme," *1985 International Symposium on Logic Programming*, pp. 110-118, IEEE, Boston, Massachusetts, July 15-18, 1985.

Moto82.

T. Moto-oka, ed., *Fifth Generation Computer Systems*, North-Holland, Tokyo, Japan, 1982.

MTSL85.

C. Mierowsky, S. Taylor, E. Shapiro, J. Levy, and M. Safra, "The Design and Implementation of Flat Concurrent Prolog," CS85-09, Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel.

Naka82.

H. Nakashima, "Prolog/KR - Language Features," *Proceedings of the First International Logic Programming Conference*, pp. 65-70, Marseille, France, September 14-17, 1982.

Naka84.

H. Nakagawa, "AND Parallel Prolog with Divided Assertion Set," *1984 International Symposium on Logic Programming*, pp. 22-28, IEEE Computer Society Press, Atlantic City, NJ, February 6-9, 1984.

NaNa87.

H. Nakashima and K. Nakajima, "Hardware Architecture of the Sequential Inference Machine: PSI-II," *Proceedings, 1987 Symposium on Logic Programming*, pp. 104–113, IEEE, San Francisco, August 31 – September 4, 1987.

NaTU84.

H. Nakashima, S. Tomura, and K. Ueda, "What is a Variable in Prolog," *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pp. 327-332, ICOT, Tokyo, Japan, November 6-9, 1984.

OKee85.

R. A. O'Keefe, "Towards an Algebra of Constructing Logic Programs," *1985 International Symposium on Logic Programming*, pp. 152-160, IEEE, Boston, Massachusetts, July 15-18, 1985.

PaSE85.

D. Patel, M. Schlag, and M. Ercegovac, "*vFP : An Environment for the Multi-Level Specification, Analysis, and Synthesis of Hardware Algorithms*," *Proceedings, Functional Programming Languages and Computer Architecture Conference*, pp. 238–255, September 16–19, 1985.

PeNa84.
L. M. Pereira and R. Nasr, "Delta Prolog: A Distributed Logic Programming Language," *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pp. 283-291, Tokyo, November 6-9, 1984.

Pere82.
L. M. Pereira, "Logic Control with Logic," *Proceedings of the First International Logic Programming Conference*, pp. 9-18, Marseille, France, September 14-17, 1982.

Quin87.
Quintus Computer Systems, Inc., *Quintus Prolog Reference Manual (Version 10)*, Mountain View, California, February 1987.

RaSh86.
K. Ramanohanarao and J. Shepherd, "A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases," *Proceedings, 3rd International Conference on Logic Programming*, pp. 569-576, London, July 14-18, 1986.

RiTh74.
D. M. Ritchie and K. Thompson, "The UNIX Timesharing System," *CACM*, vol. 17, no. 7, pp. 365-375, July 1974.

Rous75.
P. Roussel, "Prolog: Manuel de Reference et d'Utilisation," Technical Report, Groupe d'Intelligence Artificielle, Université d'Aix Mareille, 1975.

Rush81.
J. M. Rushby, "Design and Verification of Secure Systems," *Proceedings of the 8th Symposium on Operating Systems Principles*, pp. 12-20, ACM, December 1981.

SaSh86.
S. Safra and E. Shapiro, "Meta Interpreters for Real," *Proceedings IFIP-86 Congress*, North-Holland, 1986. Also available as Report CS86-11, Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel.

Sara85.
V. A. Saraswat, "Concurrent Logic Programming Languages," Thesis Proposal, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, November 1985.

SaTa84.
T. Sato and H. Tamaki, "Transformational Logic Program Synthesis," *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pp. 195-201, ICOT, Tokyo, Japan, November 6-9, 1984.

SeFu87.
H. Seki and K. Furukawa, "Notes on Transformation Techniques for Generate and Test Logic Programs," *Proceedings, 1987 Symposium on Logic Programming*, pp. 215–223, IEEE, San Francisco, August 31 – September 4, 1987.

Serg83.
M. Sergot, "A Query-the-User Facility for Logic Programming," in *Integrated Interactive Computing Systems*, ed. P. Degano and E. Sandewall, pp. 27-41, North-Holland, 1983.

Shap83a.
E. Y. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," TR-003, ICOT, Tokyo, Japan, January 1983. Also available as CS83-06, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

Shap83b.
E. Y. Shapiro, "(Lecture Notes on) The Bagel: a Systolic Concurrent Prolog Machine," TM-0031, ICOT, Tokyo, Japan, November 1983.

Shap83c.
E. Y. Shapiro, "Systems Programming in Concurrent Prolog," TR-034, ICOT, Tokyo, Japan, November 1983.

Shap84a.
E. Y. Shapiro, "Systems Programming in Concurrent Prolog," *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pp. 93-105, Salt Lake City, Utah, January 15-18, 1984.

Shap84b.
E. Y. Shapiro, "Systolic Programming: A Paradigm of Parallel Processing," *Proc. Inter. Conf. Fifth Generation Computer Systems 1984*, pp. 458-470, ICOT, Tokyo, Japan, November 6-9, 1984.

Shap86a.
E. Y. Shapiro, "On Evaluating General-Purpose Programming Languages," CS86-01, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, January 1986.

Shap86b.
E. Shapiro, "Concurrent Prolog: A Progress Report," *IEEE Computer*, vol. 19, no. 8, pp. 44-58, August 1986. Also available as Weizmann Institute of Science, Department of Computer Science Technical Report CS86-10, April 1986.

Shap87.
Concurrent Prolog: Collected Papers, M.I.T. Press, 1987.

SHHS86.
W. Silverman, M. Hirsch, A. Houri, and E. Shapiro, "The Logix System User Manual," CS86-21, Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel, 1986.

ShMi84.
E. Y. Shapiro and C. Mierowsky, "Fair, Biased, and Self-Balancing Merge Operators: Their Specification and Implementation in Concurrent Prolog," *1984 International Symposium on Logic Programming*, pp. 83-90, IEEE, Atlantic City, NJ, February 6-9, 1984.

ShSa86.
E. Y. Shapiro and M. Safra, "Fast Multiway Merge Using Destructive Operations," *New Generation Computing*, vol. 4, pp. 211–216, 1986. Also available as Tech. Report CS85-01, Department of Applied Mathematics, Weizmann Institute of Science, January 1985.

ShSh83.
A. Shafrir and E. Y. Shapiro, "Distributed Programming in Concurrent Prolog," CS83-12, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, August 1983.

ShSM86.
E. Y. Shapiro, W. Silverman, and M. Hirsch, 1986. Logix source code.

ShTa83.
E. Y. Shapiro and A. Takeuchi, "Object Oriented Programming in Concurrent Prolog," *New Generation Computing*, vol. 1, no. 1, pp. 25-48, Tokyo, 1983.

Somo87.
Z. Somogyi, "Stability of Logic Programs: how to connect don't-know nondeterministic logic programs to the outside world," Technical Report 87/11, Department of Computer Science, University of Melbourne, Parkville, Australia, September 1987.

StSh86.
L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge, Massachusetts, 1986.

SuYo86.
P. A. Subrahmanyam and J.-H. You, "FUNLOG: A Computational Model Integrating Logic Programming and Functional Programming," in *Logic Programming, functions, relations, and equations*, ed. G. Lindstrom, pp. 157–198, Prentice-Hall, 1986.

Suzu85.
N. Suzuki, "Concurrent Prolog as an Efficient VLSI Design Language," *IEEE Computer*, pp. 33–40, February 1985.

Suzu86.
N. Suzuki, "Experience with Specification and Verification of a Complex Computer Using Concurrent Prolog," in *Logic Programming and Its Applications*, ed. M. van Caneghem and D. H. D. Warren, pp. 188-207, Ablex Publishing, Norwood, New Jersey, 1986.

TaAS87.
S. Taylor, E. Av-Ron, and E. Shapiro, "A Layered Method for Process and Code Mapping," *New Generation Computing*, vol. 5, no. 2, 1987. Also available as TR CS86-17, Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel, March, 1986.

TaFu83.
A. Takeuchi and K. Furukawa, "Interprocess Communication in Concurrent Prolog," *Proceedings Logic Programming Workshop '83*, pp. 171-185, Algarve, Portugal, June 26 - July 1, 1983. Also as ICOT Technical Report TR-006, 1983.

TaFu86.
A. Takeuchi and K. Furukawa, "Parallel Logic Programming Languages," *Proceedings, 3rd International Conference on Logic Programming*, pp. 242-254, London, July 14-18, 1986.

Take86.
A. Takeuchi, "Affinity Between Meta-Interpreters and Partial Evaluation," in *Information Processing 86*, ed. H.-J. Kugler, North-Holland, 1986. Also available as ICOT TR-166.

TaSa83.
H. Tamaki and T. Sato, "A Transformation System for Logic Programs which Preserves Equivalence," TR-018, ICOT, Tokyo, Japan, August 1983.

TaSa84.
H. Tamaki and T. Sato, "Unfold/Fold Transformation of Logic Programs," *Proceedings of the Second International Logic Programming Conference*, pp. 127-138, Uppsala, Sweden, July 2-6, 1984.

TaSS87.
S. Taylor, S. Safra, and E. Shapiro, "A Parallel Implementation of Flat Concurrent Prolog," in *Concurrent Prolog: Collected Papers*, ed. E. Shapiro, pp. 375–604, M.I.T. Press, 1987.

That86.
S. M. Thatte, "Persistent Memory: Merging AI-knowledge and databases," *T.I. Engineering Journal*, vol. 3, no. 1, January-February 1986.

TMKB87.
E. D. Tribble, M. S. Miller, K. Kahn, D. G. Bobrow, and E. Shapiro, "Channels: A Generalization of Streams," *Proceedings, Fourth International Conference on Logic Programming*, pp. 839–857, Melbourne, Australia, May 25–29, 1987.

Turn84.
D. A. Turner, "Functional Programs as Executable Specifications," *Philosophical Transactions of the Royal Society of London*, vol. A 312, pp. 363-388, 1984.

TYUK84.
S. Takagi, T. Yokoi, S. Uchida, T. Kurokawa, T. Hattori, T. Chikayama, K. Sakai, and J. Tsuji, "Overall Design of SIMPOS," *Proceedings of the Second International Logic Programming Conference*, pp. 1-12, Uppsala, Sweden, July 2-6, 1984.

Uchi83.
S. Uchida, "Inference Machine: From Sequential to Parallel," *Proceedings of the 10th International Symposium on Computer Architectures*, pp. 410-416, Stockholm, June 1983. Also available as ICOT Technical Report TR-011, May 1983.

UeCh84.
K. Ueda and T. Chikayama, "Efficient Stream / Array Processing in Logic Programming Languages," *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pp. 317-326, ICOT, Tokyo, Japan, November 6-9, 1984.

UeCh85.
K. Ueda and T. Chikayama, "Concurrent Prolog Compiler on Top of Prolog," *1985 International Symposium on Logic Programming*, pp. 119-126, IEEE, Boston, Massachusetts, July 15-18, 1985.

Ueda85a.
K. Ueda, "Guarded Horn Clauses," TR-103, ICOT, Tokyo, Japan, June 1985 (Revised September 1985).

Ueda85b.
K. Ueda, "Concurrent Prolog Re-examined," TR-102, ICOT, Tokyo, Japan, November 1985.

Ueda86a.
K. Ueda, "Guarded Horn Clauses," D. Eng. Thesis, University of Tokyo, Tokyo, Japan, March 1986.

Ueda86b.
K. Ueda, "Making Exhaustive Search Programs Deterministic," *Proceedings, 3rd International Conference on Logic Programming*, pp. 270-282, London, July 14-18, 1986.

Ueda87.
K. Ueda, "Making Exhaustive Search Programs Deterministic, Part II," *Proceedings, Fourth International Conference on Logic Programming*, pp. 356–375, Melbourne, Australia, May 25–29, 1987.

UeKa83.
T. Uehara and N. Kawato, "Logic Circuit Synthesis using Prolog," *New Generation Computing*, vol. 1, no. 2, pp. 187-193, 1983.

UYYT83.
S. Uchida, M. Yokota, A. Yamamoto, K. Taki, and H. Nishikawa, "Outline of the Personal Sequential Inference Machine: PSI," *New Generation Computing*, vol. 1, no. 1, pp. 75-79, Tokyo, 1983.

Vase86.
P. Vasey, "Qualified Answers and Their Application to Transformation," *Proceedings, 3rd International Conference on Logic Programming*, pp. 425-432, London, July 14-18, 1986.

Vitt85.
J. S. Vitter, "An Efficient I/O Interface for Optical Disks," *Transactions on Database Systems*, vol. 10, no. 2, pp. 129-162, ACM, June 1985.

Warr77.
D. H. D. Warren, "Implementing Prolog – Compiling Predicate Logic Programs," Technical Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, May 1977.

Warr82.
D. H. D. Warren, "Perpetual Processes – An Unexploited Prolog Technique," *Proceedings of the Prolog Programming Environments Workshop*, Linkoping University, Sweden, March 1982.

WeSh86.
D. Weinbaum and E. Shapiro, "Hardware Description and Simulation Using Concurrent Prolog," CS86-25, Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel, September 1986.

Wise82.
M. J. Wise, "A Parallel Prolog: the Construction of a Data Driven Model," *Conference Record of the 1982 Symposium on Lisp and Functional Programming*, pp. 56-67, ACM, Pittsburgh, PA,

August 15-18, 1982.

YaAi86.

R. Yang and H. Aiso, "P-Prolog: A Parallel Logic Language Based on Exclusive Relation," *Proceedings, 3rd International Conference on Logic Programming*, pp. 255-269, London, July 14–18, 1986.

Youn85.

M. F. Young, "A Functional Language and Modular Architecture for Scientific Computing," *Proceedings, Functional Programming Languages and Computer Architecture Conference*, pp. 305–318, September 16–19, 1985.

YYTN83.

M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa, and S. Uchida, "The Design and Implementation of a Personal Inference Machine: PSI," *New Generation Computing*, vol. 1, no. 2, pp. 125–144, Tokyo, 1983.

Zani84.

C. Zaniolo, "Object-Oriented Programming in Prolog," *1984 International Symposium on Logic Programming*, pp. 265–270, IEEE, Atlantic City, New Jersey, February 6–9, 1984.

Appendix A: Introduction to Concurrent Prolog

CP (Concurrent Prolog) [Shap83a, Shap87] facilitates the expression of concurrency, communication, synchronization, and indeterminacy by a minimal extension to the basic computational model of logic programs. In CP, as opposed to Prolog, the AND- and OR-parallelism of the theoretical model of logic programs [CoKi81] is retained. A conjunctive goal can be regarded as a system of processes, a unit goal being an individual process. The state of a process is the value of a goal's arguments, and the state of a system is the union of the states of its processes. Concurrency among processes is the AND-parallelism of the theoretical model. The OR-parallel trial of candidate clauses provides each process with the ability to perform indeterminate actions. Variables shared between goals serve as the process communication mechanism. Synchronization is achieved by denoting which processes can write a variable (instantiate it to a non-variable term).

CP introduces two constructs to the model of logic programs: **read-only annotations** of variables and the **commit** operator. Read-only variable references, $X?$ where X is a variable, are used to constrain the order and pace of process reduction. Commit, denoted by '|', permits both "committed choice" and "don't care" nondeterminism [ClGr81].

A CP program is a finite set of guarded clauses. A **guarded clause** is a universally quantified axiom of the form

$$H :- G_1, \ldots, G_m \mid B_1, \ldots, B_n. \qquad m,n \geq 0$$

where the G_i's and the B_j's are atomic formulae (unit goals). H is the clause head and the G_i's form the guard. The guard may be empty, in which case the commit operator is omitted. Read-only variable references may appear within any part of a clause.

The semantics of a guarded-clause

$$H :- G \mid B.$$

are as follows. Declaratively, read-only annotations are ignored and the commit operator reads as a conjunction: H is true if G and B are true. Operationally, the clause is similar to an alternative in a guarded-command [Dijk76]. To reduce a process H' using the clause above, H and H' are unified, G is recursively reduced to the empty system, commitment is made to this clause, and H' is reduced to B. The reduction may suspend or fail at any of these steps. Unification of H and H' suspends if it requires the instantiation of variables annotated as read-only [Shap86]. It fails if H and H' are not unifiable. The reduction of the guard system G suspends if the processes in it all suspend, and fails if any of them fails. Commitment may fail if variable bindings generated by the guard computation conflict with those generated by other (concurrent) computations.

The semantics of the commit operation require that variable bindings produced by the first two steps of reduction – unification of H and H' and reduction of G – are accessible only to processes in G, or their descendants, prior to the commitment. Also, as part of commitment, all other OR-parallel attempts to reduce H' are abandoned.

Read-only variables are fundamental to CP. The **read-only operator** ('?') maps a writable (unannotated) variable to a read-only variable. If X is a writable variable, then $X?$ is called the read-only variable corresponding to X. Intuitively, a read-only variable can only be "read from" (examined or accessed), but not "written upon" (instantiated). It receives a nonvariable value only when its corresponding writable variable receives a nonvariable value. The read-only operator has no effect when applied to a term other than a writable variable. To illustrate these points, consider the following unifications (assuming X and Y are writable variables, and s and t are constants):

$X? = s$
$X? = Y?$

Read-only annotations cause suspension in each case. Unification does not suspend in

$X = Y?$
$s? = t?$.

As an aid to programming, CP contains the metalanguage predicate *otherwise*. An *otherwise* goal in a guard – it cannot appear in the body – succeeds if and when all other OR-parallel guards fail. Declaratively, it may be read as the negation of the disjunction of the guards of the sibling clauses. The *otherwise* goal can always be re-expressed as a conjunction of subgoals involving the system predicate *dif(_,_)* and a negation-as-failure primitive.

Commonly Used Concurrent Prolog Predicates

The following is a list of commonly-used CP predicates. A description is given for each.

System Predicates

otherwise

may only be used as a guard subgoal. It succeeds if and when all sibling OR-parallel guards fail [ShTa83].

wait(X)

waits until the principle functor of its argument, *X*, is determined, then terminates with success [Shap83a].

User-Definable Predicates

merge(In1,In2,Out)

computes the relation "*Out* contains the elements of *In1* and *In2*, preserving the relative order of their elements". The predicate may demonstrate various operational properties, depending on its precise definition and utilization of operational characteristics of the language implementation [Shap83a, Kusa84a, ShMi84, UeCh84, ShSa86].

receive(Msg,Strm,NStrm)

names the relation "the result of receiving *Msg* on stream *Strm* is the stream *NStrm*" [Shap83c]. It is defined by the unit clause

receive(Msg, [Msg/NStrm], NStrm?).

send(Msg,Strm,NStrm)

names the relation "the result of sending *Msg* on stream *Strm* is the stream *NStrm*" [Shap83c]. It is defined by

send(Msg, [Msg/NStrm], NStrm).

Concurrent Prolog Objects

The popular and powerful "object-oriented" programming paradigm is also available in CP. An object, in the context of CP, is a process that recursively calls itself (i.e. a perpetual process [Warr82]) and maintains an internal state by means of unshared arguments [ShTa83]. A process reduction is an instance of the object. As in other languages, CP objects communicate by message-passing. This is accomplished by unification of shared variables. Read-only variables facilitate synchronization. Objects become active on receipt of a message; otherwise, they are suspended. Typically, a CP object has a single input (request) stream. Multiple clients are handled by merging multiple streams and/or queuing. Communications are usually incomplete messages with replies transmitted through unification.

In languages not designed for the purpose, object-oriented programming often requires addition of special constructs. With CP, the realization of objects is achieved by programming style and language extensions are unnecessary.

Appendix B: Modules and Remote Procedure Calls Under Logix

Several enchancements to the language Flat Concurrent Prolog ("FCP") [MTSL85] are provided by the Logix user/development environment [SHHS86]. These are "remote procedure calls" and "modules". They are explained as follows.

A **module** is a unit of compilation under FCP and Logix. It consists of a set of procedures, optionally preceded by a module declaration. Within Logix, predicate names are not global, but local to a module. Yet, predicates defined in other modules can be given as subgoals. This is achieved with the **remote procedure call** or remote goal invocation mechanism. A procedure (predicate) may call, and be called by, procedures in other modules. Modules are loaded automatically when called and compiled automatically as needed. The syntax of a remote procedure call is

Module#Subgoal

(this indicates that the subgoal *Subgoal* is to be resolved according to the clauses in module *Module*). '#' is an infix operator.

To illustrate, consider the following example source for a module called *utilities*

```
- export( [if_ground/2] ).                % export the procedure 'if_ground(_,_)'

if_ground( Term, Resp ) :-
   freeze#freeze( Term, 0, _, N ),       % freeze the term,
                                         % with "variable counter" starting at 0
   if_ground( N?, Term, Resp ).          % check terminating counter value

if_ground( 0, true ).                    % term was ground if terminating value is 0
                                         % (i.e. not variables were encountered)
if_ground( _, false ) :- otherwise | true.
                                         % term was not ground otherwise
                                         % (terminating value is not 0)
```

Thus, in another module the goal

utilities#if_ground(Term, Response)

can be given. It will be resolved by the definition of *if_ground/2* above. The definition of *if_ground* makes use of remote procedure call facilities as well: the predicate *number_variables/4* defined in the module *freeze*.

Remote procedure calls are implemented via the stream communication capabilities of FCP. Source-to-source transformations automatically applied by the compiler change the remote goal invocations into the posting of a message on a stream bound for the named module. For example, the call of *if_ground/2* above would be transformed to

send(utilities#if_ground(Term, Response), RPCStream, NewRPCStream) .

Stream arguments for sending such messages are automatically added to clauses containing remote procedure calls. The stream arguments must also be added in the definition of (run-time) ancestor predicates.

The recipient module is subjected to source-to-source transformations as well. An "outer" process is added which accepts messages (the remote procedure calls) on an input stream. The process then invokes appropriate goals for which clauses are given in the module. To illustrate, the *utilities* module above may have added to it the definition of the perpetual process *serve*:

```
serve( [if_ground(Term,Response)|RPCStrm] ) :-
   if_ground( Term, Response ),          % invoke goal
   serve( RPCStrm? ).                    % wait for another RPC
serve( [UnrecognizedRPC|RPCStrm] ) :-
   otherwise |
   error#error( utilities, not_recognized, UnrecognizedRPC ),
   serve( RPCStrm? ).
```

The main computation preformed by the *utilities* is then the (perpetual) *serve* process. It accepts messages of the form *if_ground(_,_)* and invokes *if_ground/2* subgoals to respond to the messages.

All transformations necessary for these features are performed by a preliminary stage of the compiler. The overheads of the mechanisms are transparent to the user and programmer. The transformations involved maintain the equivalence of programs.

Appendix C: Enhanced File System Kernel

Client interface to file system kernel.

This module is called *fs_interface*. It checks for correct form of client requests. It suspends any requests in which the file name is not ground.

```
- export( affirm/4, modify/4, replace/4, access/3, revoke/3, inventory/2 ).
/*
 * the FCP (under LOGIX) compiler adds a higher-level predicate which
 * receives affirm, modify, etc. messages and issues the appropriate
 * goal call
 */

/*
 * main file system kernel process is called 'fs'
 */

/*
 * for an 'affirm' request, start a concurrent process which
 * waits until storage of the file is indicated.
 * <FileContent> is an arbitrary term.  It is frozen when the third
 * argument of the 'affirm' message becomes ground.
 * However, if 'FreezeOn' is bound to 'abort', the state of
 * the file system is unaffected by the 'affirm' request.
 */
affirm( FileName, FileContent, FreezeOn, Response ) :-
     ground( immediate, FileName, FileNameGnd ),
     new_file( FileNameGnd?, FileName, FileContent, FreezeOn, Response ).

/*
 * save user trouble of issuing separate 'access'/'revoke'/'affirm'
 * requests by supporting 'modify' and 'replace' requests.
 * The difference between the two is that with 'modify', 'FileContent'
 * is unified with the previous content; with 'replace', previous
 * content is discarded.
 */
modify( FileName, FileContent, FreezeOn, Response ) :-
     fs_interface#revoke(FileName,FileContent,AccessResponse),
     replace( AccessResponse?, FileName, FileContent, FreezeOn, Response ).

replace( FileName, FileContent, FreezeOn, Response ) :-
     fs_interface#revoke(FileName,IgnoreFileContent,AccessResponse),
     replace( AccessResponse?, FileName, FileContent, FreezeOn, Response ).

access( FileName, FileContent, Response ) :-
     ground( immediate, FileName, FileNameGnd ),
     forward( FileNameGnd?, access(FileName,FileContent,Response) ).

revoke( FileName, FileContent, Response ) :-
     ground( immediate, FileName, FileNameGnd ),
     forward( FileNameGnd?, revoke(FileName,FileContent,Response) ).

inventory( FileNamePattern, ListOfNames ) :-
     fs#inventory(FileNamePattern,ListOfNames).

/*
 * auxiliary predicates
 */

/*
 * forward the message if first argument of call is 'true'
 */
```

```
forward( true, Msg ) :-
     fs#Msg.
forward( false, Msg ) :-
     screen#output(fs,file_name_not_ground( Msg )).

/*
 * need two types of 'ground'
 *   -    one which waits until a term is ground
 *   -    one which returns 'true' or 'false' immediately
 */
ground( wait, Term, WhenGnd ) :-
     utilities#ground(Term,WhenGnd).
ground( immediate, Term, IsGnd ) :-
     utilities#if_ground(Term,IsGnd).

/*
 * predicate to assist with creation of new file
 */
new_file( false, FileName, FileContent, FreezeOn, false ) :-
     screen#output(fs,problem_with_file_name( FileName )).
new_file( true, FileName, FileContent, FreezeOn, Response ) :-
     wait_for_freeze( FreezeOn?, Response,
          affirm(FileName,FileContent,Response) ).

/*
 * predicate to assist with replacement of a file
 */
replace( false, FileName, FileContent, FreezeOn, false ).
replace( true, FileName, FileContent, FreezeOn, Response ) :-
     wait_for_freeze( FreezeOn?, Response,
          replace(FileName,FileContent,Response) ).

/*
 * send a message to 'fs' when first argument becomes instantiated
 */
send( FreezeNow, Message ) :-
     wait( FreezeNow ) | fs#Message.

wait_for_freeze( abort, abort, Msg ).
wait_for_freeze( FreezeOn, Response, Msg ) :-
     otherwise |
     ground( wait, FreezeOn, FreezeNow ),
     send( FreezeNow?, Msg ).
```

Main file system kernel process.

This module is called *fs*. It actually manages the files.

```
/*
 * this program defines a perpetual process.  Due to the characteristics
 * of LOGIX, the first predicate invoked in this case is 'monitor/1'
 */
monitor( Input ) :-
     fs( Input? ).

fs( [init(fs)|Input] ) :-
     storage_medium#last_id(LastId,IgnoreResponse1),
     retrieve( LastId?, FileSysHistory, IgnoreResponse2 ),
     init_state( LastId?, FileSysHistory?, UcrDB ),
     fs( Input?, UcrDB? ).
fs( [] ).

/*
 * main predicate which is recursively called to process each
 * request
 */
fs( [affirm(Name,Term,Response)|Input], UcrDB ) :-
     find( Name, UcrDB, _, FindResponse ),
     fs( [affirm( Name, Term, Response )|Input], FindResponse?, UcrDB ).
fs( [revoke(Name,Term,Response)|Input], db( IdCurrentStore, FileSys ) ) :-
     find( Name, FileSys, NewFileList, FindResponse ),
     fs( [revoke( Name, Term, Response )|Input], FindResponse?,
          db( IdCurrentStore, NewFileList? ) ).
/*
 *   'inventory' and 'access' can run asynchronously,
 *   but 'affirm' and 'revoke' cannot
 */
fs( [access( Name, Term, Response )|Input], UcrDB ) :-
     find( Name, UcrDB, NewFileSys, FindResponse ),
     fs( [access( Name, Term, Response )], FindResponse?, UcrDB ),
     fs( Input?, UcrDB ).
fs( [inventory(FileNamePattern,ListOfNames)|Input], UcrDB ) :-
     inventory( FileNamePattern, UcrDB, ListOfNames[] ),
     fs( Input?, UcrDB ).
fs( [], UcrDB ).
fs( [Request|Input], UcrDB ) :-
     otherwise |
     screen#output(fs,dont_understand(Request)),
     fs( Input?, UcrDB ).

/*
 * different form of 'fs' predicate with 'FindResponse' inserted
 * as new second argument
 */
fs( [affirm( Name, Term, Response )|Input], false, db( IdCurrentStore, FileSys ) ) :-
     store( gnd, UcrOfTerm, Term, true ),
     NewFileSys = [(Name,UcrOfTerm?)|FileSys],
     store( UcrOfTerm, NewStoreId, [NewFileSys|IdCurrentStore], Response ),
     fs( Input?, db( NewStoreId?, NewFileSys? ) ).
fs( [affirm( Name, Term, false )|Input], FindResponse, UcrDB ) :-
     otherwise |
     screen#output(fs,file_already_exists(Name)),
     fs( Input?, UcrDB ).
fs( [revoke( Name, Term, false )|Input], false, UcrDB ) :-
     screen#output(fs,file_not_found(Name)),
     fs( Input?, UcrDB ).
```

```
fs( [revoke( Name, Term, Response )|Input], ucr( UCR ),
    db( IdCurrentStore, NewFileSys ) ) :-
     retrieve( UCR?, Term, true ),
     store( NewFileSys, NewStoreId, [NewFileSys?|IdCurrentStore], Response ),
     fs( Input?, db( NewStoreId?, NewFileSys? ) ).
fs( [access( Name, Term, Response )], ucr( UCR ), UcrDB ) :-
     retrieve( UCR, Term, Response ).
fs( [access( Name, Term, false )], false, UcrDB ) :-
     screen#output(fs,file_not_found(Name)).

/*
 * auxilliary predicates
 */

/*
 * predicate to try to find file entry (given a file name)
 * in the current file system data structure
 */
find( FileName, [(FileName,UCR)|FileSys], FileSys, ucr( UCR ) ).
find( FileName, [], [], false ).
find( FileName, [DifferentFile|FileSys], [DifferentFile|NewFileSys], FindResponse ) :-
     otherwise |
     find( FileName, FileSys?, NewFileSys, FindResponse ).
/*
 *   Allow predicates which make use of this predicate to optimize a
 *   bit on the form of their own clauses (i.e. no repeated
 *   destruction/construction of UcrDB)
 */
find( FileName, db( IdCurrentStore, FileSys ), NewFileSys, Response ) :-
     find( FileName, FileSys, NewFileSys, Response ).

init_state( IdMostRecentStore, [], db( IdMostRecentStore, [] ) ).
init_state( IdMostRecentStore, [FileSys|IdNextRecentStore],
     db( IdMostRecentStore, FileSys? ) ).
init_state( IdMostRecentStore, ProportsToBeFileStore, db( EndOfList?, [] ) ) :-
     otherwise |
     storage_medium#assert(EndOfList,[],IgnoreResponse),
     screen#output(fs,corrupt_file_sys(ProportsToBeFileStore)),
     screen#output(fs,reinitializing_with_null_fs).

/*
 * given a file name pattern and the (current) file system content,
 * construct a list of (unifying) file names
 */
inventory( FileNamePattern, [], ListOfNames\ListOfNames ).
inventory( FileNamePattern, [File|FileSysDB], ListOfNamesH\ListOfNamesT ) :-
     library#copy(FileNamePattern,Copy,Done),
     inventory_match( Done?, Copy, File, ListOfNamesH\RemListOfNames ),
     inventory( FileNamePattern, FileSysDB, RemListOfNames\ListOfNamesT ).
/*
 *   Allow predicates which make use of this predicate to optimize a
 *   bit on the form of their own clauses (i.e. no repeated
 *   destruction/construction of UcrDB)
 */
inventory( FileNamePattern, db( IdCurrentStore, FileSys ), ListOfNames ) :-
     inventory( FileNamePattern, FileSys, ListOfNames ).

/*
 * when copy of file name pattern is made, see if it unifies with file
 * name of next file (file names do not have to be copied before
 * unification as it is ensured that are ground earlier)
 */
```

```
inventory_match( done, FileName, (FileName,FileContent), [FileName|List]\List ).
inventory_match( done, FileName, HasDifferentName, List\List ) :-
     otherwise | true.

/*
 * send a message to 'storage_medium' to query 'Ucr' when
 * 'Ucr' becomes instantiated
 */
retrieve( Ucr, Term, Response ) :-
     wait( Ucr ) | storage_medium#query( Ucr,Term, Response ).

/*
 * send a message to 'storage_medium' to assert 'Term'
 * when 'WaitFor' becomes bound
 */
store( WaitFor, Ucr, Term, Response ) :-
     wait( WaitFor ) |
     storage_medium#assert(Ucr,Term,Response).
```

Appendix D: Further Extended File System Kernel

Client interface to file system kernel.

This module is called *fs_interface*. It checks for correct form of client requests. It suspends any requests in which the file name is not ground. It is very similar to the interface module in Appendix C, except that

- it also supports abbreviated forms of client requests;
- in requests having an argument *FreezeOn*, the variable can be bound to *abort* to "cancel" the request.

```
- export( [ access/2, access/3,
         affirm/2, affirm/3, affirm/4,
         modify/2, modify/3, modify/4,
         replace/2, replace/3, replace/4,
         revoke/2, revoke/3,
         inventory/1, inventory/2 ] ).
/*
 * the FCP (under LOGIX) compiler adds a higher-level predicate which
 * receives affirm, modify, etc. messages and issues the appropriate
 * goal call
 *
 * note that abbreviated forms of most requests are available.
 */

access( FileName, FileContent, Response ) :-
       utilities#if_ground( FileName, FileNameGnd ),
       forward( FileNameGnd?, access(FileName,FileContent,Response), Response ).

access( FileName, FileContent ) :-
       access( FileName, FileContent, Response ),
       check_for_error( access( FileName, FileContent ), Response? ).

/*
 * for an 'affirm' request, start a concurrent process which
 * waits for until storage of the file is indicated.
 * <FileContent> is an arbitrary term.  It is frozen when the third
 * argument of the 'affirm' message becomes ground.
 * However, if 'FreezeOn' is bound to 'abort', the state of
 * the file system is unaffected by the 'affirm' request.
 */
affirm( FileName, FileContent, FreezeOn, Response ) :-
       utilities#if_ground( FileName, FileNameGnd ),
       new_file( FileNameGnd?, FileName, FileContent, FreezeOn, Response ).

affirm( FileName, FileContent, FreezeOn ) :-
       affirm( FileName, FileContent, FreezeOn, Response ),
       check_for_error( affirm( FileName, FileContent, FreezeOn ), Response? ).

affirm( FileName, FileContent ) :-
       affirm( FileName, FileContent, freeze_now, Response ),
       check_for_error( affirm( FileName, FileContent ), Response? ).

check_for_error( Function, true ).
check_for_error( Function, false ) :-
       error#error( storage_medium, could_not, Function ).

forward( true, Msg, Response ) :-
       fs#Msg.
forward( false, Msg, false ) :-
       screen#output(fs,file_name_not_ground( Msg )).
```

```
inventory( FileNamePattern, ListOfNames ) :-
       fs#inventory( FileNamePattern, ListOfNames ).

inventory( FileNamePattern ) :-
       inventory( FileNamePattern, ListOfNames ),
       screen#output( wait(ListOfNames), inventory(FileNamePattern), ListOfNames ).
/*
 * save user trouble of issuing separate 'access'/'revoke'/'affirm'
 * requests by supporting 'modify' and 'replace' requests.
 * The difference between the two is that with 'modify', 'FileContent'
 * is unified with the previous content; with 'replace', previous
 * content is discarded.  Actual previous content of the file is not
 * removed until 'FreezeOn' becomes ground.
 */
modify( FileName, FileContent, FreezeOn, Response ) :-
       access( FileName, FileContent, AccessResponse ),
       replace( AccessResponse?, FileName, FileContent, FreezeOn, Response ).

modify( FileName, FileContent, FreezeOn ) :-
       modify( FileName, FileContent, FreezeOn, Response ),
       check_for_error( modify( FileName, FileContent, FreezeOn ), Response? ).

modify( FileName, FileContent ) :-
       modify( FileName, FileContent, freeze_now, Response ),
       check_for_error( modify( FileName, FileContent ), Response? ).

new_file( false, FileName, FileContent, FreezeOn, false ) :-
       screen#output(fs,problem_with_file_name( FileName )).
new_file( true, FileName, FileContent, FreezeOn, Response ) :-
       utilities#ground( FreezeOn?, FreezeNow ),
       wait_for_freeze( FreezeNow?, Response,
                     affirm( FileName, FileContent, Response ) ).

revoke( FileName, FileNameContent, Response ) :-
       utilities#if_ground( FileName, FileNameGnd ),
       forward( FileNameGnd?, revoke( FileName, FileNameContent, Response ),
              Response ).

revoke( FileName, FileContent ) :-
       revoke( FileName, FileContent, Response ),
       check_for_error( revoke( FileName, FileContent ), Response? ).

replace( FileName, FileContent, FreezeOn, Response ) :-
       access( FileName, IgnoreFileContent, AccessResponse ),
       replace( AccessResponse?, FileName, FileContent, FreezeOn, Response ).

replace( FileName, FileContent, FreezeOn ) :-
       replace( FileName, FileContent, FreezeOn, Response ),
       check_for_error( replace( FileName, FileContent, FreezeOn ), Response? ).

replace( FileName, FileContent ) :-
       replace( FileName, FileContent, freeze_now, Response ),
       check_for_error( replace( FileName, FileContent ), Response? ).

replace( false, FileName, FileContent, FreezeOn, false ).
replace( true, FileName, FileContent, FreezeOn, Response ) :-
       utilities#ground( FreezeOn?, FreezeNow ),
       wait_for_freeze( FreezeNow?, Response,
              replace( FileName, FileContent, Response ) ).

/*
 * 'FreezeOn' variable in 'affirm', 'modify', and 'replace'
```

```
 * requests can also be 'abort', which has the obvious, expected
 * effect.
 */
wait_for_freeze( abort, abort, Msg ) :-
      screen#output( fs, request_aborted( Msg ) ).
wait_for_freeze( FreezeOn, Response, Msg ) :-
      otherwise | fs#Msg.

/*
 * A possible modification is to relax the restriction on ground
 * file names for 'access' requests.  However, it must be carefully
 * considered if the restriction should be dropped in the case of
 * 'modify' and 'replace'.
 */
```

Generic file system kernel agent.

This is the main module which actually manages the files. The root agent is called *fs* - the others can only be reached through streams internal to the agent hierarchy.

```
/*
 * The first predicate invoked is 'monitor/1'
 */
monitor( Input ) :-
     fs( Input? ).

add_dir_entry( {UcrDB,StrmDB}, NewName, NewUcr,
     {[d(NewName,NewUcr)|UcrDB],[d(NewName,ucr(NewUcr))|StrmDB]} ).

add_ucr_entry( {UcrDB,StrmDB}, Entry, {[Entry|UcrDB],StrmDB} ).

add_new_strm( {UcrDB,StrmDB}, DirName, NewStrm, UnaffectedStrms,
     {UcrDB,[d(DirName,NewStrm)|UnaffectedStrms]} ).

close_strms( [] ).
close_strms( [d(DirName,strm([]))|StrmDB] ) :- close_strms( StrmDB? ).
close_strms( [d(DirName,ucr(UCR))|StrmDB] ) :- close_strms( StrmDB? ).

find( {HigherOrderPart,RestOfName}, {UcrDB,StrmDB}, Response ) :-
     find_strm( HigherOrderPart?, RestOfName?, StrmDB?, PreviousStrms,
          PreviousStrms, Response ).
find( Name, {UcrDB,StrmDB}, Response ) :-
     otherwise |
     find_ucr( Name, f, UcrDB?, OtherEntries, OtherEntries, Response ).

find_strm( DirName,
     RestOfName,
     [d(DirName,Strm)|StrmDB],
     UnaffectedStrms,
     StrmDB,
     forward( RestOfName, d(DirName,Strm), UnaffectedStrms ) ).
find_strm( DirName,
     RestOfName,
     [],
     UnaffectedStrms,
     [],
     not_found( DirName ) ).
find_strm( DirName,
     RestOfName,
     [Entry|StrmDB],
     UnaffectedStrms,
     [Entry|More],
     Response
) :-
     otherwise |
     find_strm( DirName, RestOfName, StrmDB?, UnaffectedStrms,
          More, Response ).

find_ucr_of_dir( Name, { UcrDB, StrmDB}, Response ) :-
     find_ucr( Name, d, UcrDB?, OtherEntries, OtherEntries, Response ).

find_ucr( Name,
     Type,
     [{Type,Name,UCR}|UcrDB],
     OtherEntries,
     UcrDB,
     found( UCR, OtherEntries ) ).
```

```
find_ucr( Name,
     Type,
     [],
     OtherEntries,
     [],
     not_found( Name ) ).
find_ucr( Name,
     Type,
     [Entry|UcrDB],
     OtherEntries,
     [Entry|More],
     Response
) :-
     otherwise |
     find_ucr( Name, Type, UcrDB?, OtherEntries, More, Response ).

forward( Msg,
     ChildInput,
     d( DirName, ucr( UCR ) ),
     cache( ThisDir, ThisDirUcr, ParentStrm ),
     strm( WorkerIn ),
     NewChildInput
) :-
     fs( [init( DirName, UCR, StrmToThisProcess ),Msg|WorkerIn?]
                          % new instance of the agent
     stream#merger( [merge(StrmToThisProcess?)|ChildInput?],
               NewChildInput ).
forward( Msg,
     Input,
     d( DirName, strm( StrmToChild ) ),
     Cache,
     strm( NewStrmToChild ),
     Input
) :-
     send( Msg, StrmToChild, NewStrmToChild ).

/*
 * clauses to start up master agent or slave-copy of agent
 */

/*
 * This clause is used by each slave as it starts up
 */
fs( [init(DirectoryName,DirectoryUcr,ParentStrm)|ParentInput] ) :-
     retrieve( DirectoryUcr?, DirectoryTerm, IgnoreResponse ),
     init_internal_db( DirectoryTerm?, IntDB ),
     fs( [], ParentInput?, cache(DirectoryName?,DirectoryUcr?,ParentStrm), IntDB? ).
/*
 * This clause initializes the process at the top of the process tree
 */
fs( [] ).
fs( [init(fs)|Input] ) :-
     otherwise |
     storage_medium#last_id( LastId ),
     retrieve( LastId?, FileSysHistory, IgnoreResponse ),
     init_fs( LastId?, FileSysHistory?, IntDB, IdMostRecentStore ),
     fs( [], Input?, cache('_root', IdMostRecentStore?, [] ), IntDB? ).
/*
 * slaves are only started up as they are needed -
 * they are not created in advance.  When started the input stream
 * from children is (initially) null.
 */
```

```
/*
 * clauses to handle special requests internal to
 * the tree of agent processes
 */
fs(  [update_directory( DirName, NewUcr )|ChildInput],
     ParentInput,
     Cache,
     IntDB
) :-
     find_ucr_of_dir( DirName?, IntDB, Result ),
     update_ucr_list( DirName?, IntDB, Result?, NewUcr?, NewIntDB ),
     update_directory( NewUcr?, Cache, NewIntDB?, NewCache ),
     fs( ChildInput?, ParentInput, NewCache?, NewIntDB? ).
/*
 * clauses to actually respond to requests
 */
fs(  ChildInput,
     [affirm(Name,Term,Response)|ParentInput],
     Cache,
     IntDB
) :-
     find( Name?, IntDB, FindResponse ),
     fs(  ChildInput,
          [affirm( Name, Term, Response )|ParentInput],
          FindResponse?,
          Cache,
          IntDB ).
fs(  ChildInput,
     [revoke(Name,Term,Response)|ParentInput],
     Cache,
     IntDB
) :-
     find( Name?, IntDB, FindResponse ),
     fs(  ChildInput,
          [revoke( Name, Term, Response )|ParentInput],
          FindResponse?,
          Cache,
          IntDB ).
fs(  ChildInput,
     [replace(Name,Term,Response)|ParentInput],
               % 'replace' supported to make the 'revoke'/'affirm'
     Cache,          % pair of transactions more efficient by eliminating
     IntDB     % the need to generate two versions of the file
) :-           % system content for just the change of one file
     find( Name?, IntDB, FindResponse ),
     fs(  ChildInput,
          [replace( Name, Term, Response )|ParentInput],
          FindResponse?,
          Cache,
          IntDB ).
fs(  ChildInput,
     [access( Name, Term, Response )|ParentInput],
     Cache,
     IntDB
) :-
     find( Name?, IntDB, FindResponse ),
     fs(  ChildInput,
          [access( Name, Term, Response )|ParentInput],
          FindResponse?,
          Cache,
          IntDB ).
fs(  ChildInput,
```

```
     [inventory(NamePattern,ListOfNames)|ParentInput],
     Cache,
     IntDB
) :-
     locate( NamePattern, IntDB, Response ),
     fs(  ChildInput,
          [inventory(NamePattern,ListOfNames)|ParentInput],
          Response?,
          Cache,
          IntDB ).
fs(  ChildInput,
     [],
     cache(DirectoryName,DirectoryUcr,[]),
     {UcrDB,StrmDB}
) :-
     close_strms( StrmDB? ).
fs(  ChildInput,
     [Request|ParentInput],
     Cache,
     IntDB
) :-
     otherwise |
     screen#output(fs,dont_understand(Request)),
     fs( ChildInput, ParentInput?, Cache, IntDB ).

/*
 * different form of 'fs' predicate with 'FindResponse' inserted
 * as new second argument
 */

fs(  ChildInput,
     [{Request,Name,Term,Response}|ParentInput],
     forward( RestOfName, d( DirName, Strm ), UnaffectedStrms ),
     Cache,
     IntDB
) :-
     forward( {Request,RestOfName,Term,Response}, ChildInput,
          d( DirName, Strm? ), Cache, NewStrm, NewChildInput ),
     add_new_strm( IntDB, DirName?, NewStrm?, UnaffectedStrms?, NewIntDB ),
     fs( NewChildInput?, ParentInput?, Cache, NewIntDB? ).
fs(  ChildInput,
     [{Request,Name,Response}|ParentInput],
     forward( RestOfName, d( DirName, Strm ), UnaffectedStrms ),
     Cache,
     IntDB
) :-
     forward( {Request,RestOfName,Response}, ChildInput,
          d( DirName, Strm? ), Cache, NewStrm, NewChildInput ),
     add_new_strm( IntDB, DirName?, NewStrm?, UnaffectedStrms?, NewIntDB ),
     fs( NewChildInput?, ParentInput?, Cache, NewIntDB? ).
fs(  ChildInput,
     [affirm( Name, Term, Response )|ParentInput],
     not_found( Name ),   % if 'Name' in request is same as search
                          % 'Name', then a ucr was sought
     Cache,
     IntDB
) :-
     storage_medium#assert( UcrOfTerm, Term, Response ),
     add_ucr_entry( IntDB, f(Name?,UcrOfTerm?), NewIntDB ),
     update_directory( UcrOfTerm?, Cache, NewIntDB?, NewCache ),
     fs( ChildInput, ParentInput?, NewCache?, NewIntDB? ).
fs(  ChildInput,
```

```
     [affirm( Name, Term, Response )|ParentInput],
     not_found( SearchName ), % if 'Name' in request is not the
                         % same as search 'Name', then a
                         % stream was sought, but was not
     Cache,                   % found.  Therefore, need to
     IntDB                    % create a new directory at this
) :-                          % level
     otherwise |              % create a directory with nothing in it
     storage_medium#assert( UcrOfDir, [], true ),
     add_dir_entry( IntDB, SearchName?, UcrOfDir?, NewIntDB ),
                    % don't bother updating directory at this
                    % point as request to do so will come back
                    % up from descendants once final instance
                    % of server is created for this new file
     fs(  ChildInput,
          [affirm(Name,Term,Response)|ParentInput],
          Cache,              % reprocess the request
          NewIntDB? ).        % (this time it should be forwarded)
fs(  ChildInput,
     [affirm( Name, Term, false )|ParentInput],
     found( UCR, OtherEntries ),
     Cache,
     IntDB
) :-
     screen#output(fs,file_already_exists(Name)),
     fs( ChildInput, ParentInput?, Cache, IntDB ).
fs(  ChildInput,
     [revoke( SearchName, Term, false )|ParentInput],
     not_found( SearchName ),
     Cache,
     IntDB
) :-
     screen#output(fs,file_not_found(SearchName)),
     fs( ChildInput, ParentInput?, Cache, IntDB ).
fs(  ChildINput,
     [revoke( Name, Term, Response )|ParentInput],
     found( UCR, OtherEntries ),
     Cache,
     IntDB
) :-
     retrieve( UCR, Term, Response ),
     remove_entry( IntDB, found( UCR, OtherEntries ), NewIntDB ),
     update_directory( OtherEntries?, Cache, NewIntDB?, NewCache ),
     fs( ChildInput, ParentInput?, NewCache?, NewIntDB? ).
fs(  ChildInput,
     [replace( SearchName, Term, false )|ParentInput],
     not_found( SearchName ),
     Cache,
     IntDB
) :-
     screen#output(fs,file_not_found(SearchName)),
     fs( ChildInput, ParentInput?, Cache, IntDB ).
fs(  ChildInput,
     [replace( Name, Term, Response )|ParentInput],
     found( UCR, OtherEntries ),
     Cache,
     IntDB
) :-
     remove_entry( IntDB, found( UCR, OtherEntries ), NewIntDB ),
     fs(  ChildInput,
          [affirm(Name,Term,Response)|ParentInput],
          not_found( Name ),
```

```
          Cache,
          NewIntDB? ).
fs(  ChildInput,
     [access( Name, Term, Response )|ParentInput],
     found( UCR, OtherEntries ),
     Cache,
     IntDB
) :-
     retrieve( UCR, Term, Response ),
     fs( ChildInput, ParentInput?, Cache, IntDB ).
fs(  ChildInput,
     [access( Name, Term, false )|ParentInput],
     not_found( SearchName ),
     Cache,
     IntDB
) :-
     screen#output(fs,file_not_found(SearchName)),
     fs( ChildInput, ParentInput?, Cache, IntDB ).
fs(  ChildInput,
     [inventory(NamePattern,[])|ParentInput],
     not_found( SearchPattern ),
     Cache,
     IntDB
) :-
     fs( ChildInput, ParentInput?, Cache, IntDB ).
fs(  ChildInput,
     [inventory(NamePattern,ListOfNames)|ParentInput],
     found( Ignore ),
     Cache,
     IntDB
) :-
     inventory( NamePattern, IntDB, ListOfNames ),
     fs( ChildInput, ParentInput?, Cache, IntDB ).

init_fs( IdMostRecentStore,
     [],
     IntDB,
     IdMostRecentStore
) :-
     init_internal_db( [], IntDB ).
init_fs( IdMostRecentStore,
     [fs(FileSys)|IdNextRecentStore],
     IntDB,
     IdMostRecentStore
) :-
     init_internal_db( FileSys?, IntDB ).
init_fs( ProportsToBeIdMostRecentStore,
     ProportsToBeFileStore,
     IntDB,
     NewFileSysId
) :-
     otherwise |
     storage_medium#assert( NewFileSysId, [], true ),
     screen#output(fs,corrupt_file_sys(ProportsToBeFileStore)),
     screen#output(fs,reinitializing_with_null_fs),
     init_internal_db( [], IntDB ).

init_internal_db( DirContent, {DirContent,StrmDB} ) :-
     init_strm_db( DirContent, StrmDB ).

init_strm_db( [], [] ).
init_strm_db( [f(FileName,UCR)|DirContent], StrmDB ) :-
```

```
     init_strm_db( DirContent?, StrmDB ).
init_strm_db( [d(DirectoryName,UCR)|DirContent],
     [d(DirectoryName,ucr(UCR))|StrmDB]
) :-
     init_strm_db( DirContent?, StrmDB ).
init_strm_db( [SomethingUnexpected|DirContent], StrmDB ) :-
     otherwise |
     screen#output(fs,unexpected_file_sys_entry(SomethingUnexpected)),
     init_strm_db( DirContent?, StrmDB ).
init_strm_db( SomethingUnexpected, [] ) :-
     otherwise |
     screen#output(fs,unexpected_directory_content(SomethingUnexpected)).

inventory( FileNamePattern, [], ListOfNames\ListOfNames ).
inventory( FileNamePattern, [Entry|UcrDB], ListOfNamesH\ListOfNamesT ) :-
     library#copy(FileNamePattern,Copy,Done),
     inventory_match( Done?, Copy, Entry, ListOfNamesH\RemListOfNames ),
     inventory( FileNamePattern, UcrDB, RemListOfNames\ListOfNamesT ).
/*
 *   Allow predicates which make use of this predicate to optimize a
 *   bit on the form of their own clauses (i.e. no repeated
 *   destruction/construction of IntDB)
 */
inventory( FileNamePattern, {UcrDB,StrmDB}, ListOfNames ) :-
     inventory( FileNamePattern, UcrDB?, ListOfNames\[] ).

inventory_match( done, SameName, d(SameName,Ucr), [{SameName,Var}|List]\List ).
inventory_match( done, SameName, f(SameName,Ucr), [SameName|List]\List ).
inventory_match( done, Name, HasDifferentName, List\List ) :-
     otherwise | true.

locate( NamePattern, {UcrDB,StrmDB}, found( Dummy ) ) :-
     var( NamePattern ) | true.
locate( NamePattern, {UcrDB,StrmDB}, Response ) :-
     otherwise |
     locate( NamePattern, UcrDB?, StrmDB?, Response ).

/*
 * this form very similar to find/3, except that find_strm/6
 * and find_ucr/5 are not called with "name" args read-only
 */
locate( {HigherOrderPart,RestOfName}, UcrDB, StrmDB, Response ) :-
     find_strm( HigherOrderPart, RestOfName, StrmDB?, PreviousStrms,
          PreviousStrms, Response ).
locate( NamePattern, UcrDB, StrmDB, Response ) :-
     otherwise |
     find_ucr( NamePattern, AnyType, UcrDB?, OtherEntries,
          OtherEntries, Response ).

remove_entry( {UcrDB,StrmDB}, found( UCR, OtherEntries ), {OtherEntries,StrmDB} ).

/*
 * send a message to 'storage_medium' to query 'Ucr' when
 * 'Ucr' becomes instantiated
 */
retrieve( Ucr, Term, Response ) :-
     wait( Ucr ) | storage_medium#query( Ucr,Term, Response ).

send( Msg, [Msg|Strm], Strm ).

update_directory( WaitFor,
     cache( '_root', IdRecentStore, ParentStrm ),
```

```
     {UcrDB, StrmDB},
     cache( '_root', IdMostRecentStore, ParentStrm )
) :-
     wait( WaitFor ),
     wait( IdRecentStore ) |
     storage_medium#assert( IdMostRecentStore, [fs(UcrDB)|IdRecentStore], true ).
update_directory( WaitFor,
     cache( DirName, DirectoryUcr, ParentStrm ),
     {UcrDB,StrmDB},
     cache( DirName, NewDirectoryUcr?, NewParentStrm )
) :-
     otherwise, wait( WaitFor ) |
     storage_medium#assert( NewDirectoryUcr, UcrDB?, true ),
     send(update_directory( DirName,NewDirectoryUcr? ),
          ParentStrm,
          NewParentStrm ).

update_ucr_list( DirName,
     {UcrDB,StrmDB},
     found( OldUcr, RemUcrDB ),
     NewUcr,
     {[d(DirName,NewUcr)|RemUcrDB],StrmDB} ).
```