AD-A105 661 SYSTEMS CONTROL INC PALO ALTO CA COMPUTER SCIENCE DEPT F/G 9/2
RESEARCH ON KNOWLEDGE BASED PROGRAMMING AND ALGORITHM DESIGN.(U)
AUG 81 C GREEN N00014-79-C-0127
UNCLASSIFIED SCI-ICS-L-81-5 NL

1 OF 2
AD
A105661

LEVEL II

12

SCI.ICS.L.81.5

RESEARCH ON KNOWLEDGE BASED PROGRAMMING AND ALGORITHM DESIGN

Cordell Green
Principal Investigator
Systems Control, Inc.

August 1981

FINAL REPORT

Prepared for

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22217

under

Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217

DARPA Order 3687 ✓
Contract N00014-79-C-0127 ✓
27 November 1978 - 26 November 1979

and

DARPA Order 3828
Contract N00014-80-C-0045 new
22 October 1979 - 30 April 1981

The views and conclusions in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of SCI, DARPA, ONR, or the US government.

AD A105661

DTIC FILE COPY

DTIC
SELECTE
OCT 16 1981
S E D

This document has been approved for public release and sale; its distribution is unlimited.

81 9 08 124

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(14) **REPORT DOCUMENTATION PAGE** — READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: SCI-ICS-L-81-5	2. GOVT ACCESSION NO.: AD-A105661	3. RECIPIENT'S CATALOG NUMBER: (9)
4. TITLE (and Subtitle): RESEARCH ON KNOWLEDGE BASED PROGRAMMING AND ALGORITHM DESIGN.		5. TYPE OF ~~REPORT & PERIOD COVERED~~: ~~Combined~~ Final Technical Report, 27 Nov 1978-31 Aug 1981
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Cordell Green, Principal Investigator (10)	(15)	8. CONTRACT OR GRANT NUMBER(s): N00014-79-C-0127 and N00014-80-C-0045
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Systems Control, Incorporated, 1801 Page Mill Road, Palo Alto, CA 94304		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ARPA Order Nos. 3687 ✓ and 3828
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, Attn: Program Management/MIS, 1400 Wilson Boulevard Arlington, VA 22209	(11)	12. ~~REPORT DATE~~: August 1981
		13. NUMBER OF PAGES: 112
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, 800 N. Quincy Street, Arlington, VA 22217, Attn: Codes 437, 200, 455, 458	(12) 121	15. SECURITY CLASS. (of this report): Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

program synthesis
programming environments
automatic programming
artificial intelligence
expert systems
knowledge-based programming
algorithm design

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes research in knowledge-based programming and algorithm design during the period 1978 to 1981. The report includes a brief discussion of the PSI system that was completed in 1979, and provides a guide to the literature about PSI.

The emphasis of the report is on the more recent research on the CHI knowledge-based programming system and on the closely related Algorithm Design project.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

412200

(BLOCK 20 CONCLUDED)
Documentation on several aspects of these projects is available for the first time in this report, and much more documentation will be forthcoming during the next contract period.

The object of our research is the codification of programming knowledge and the creation of computer systems that incorporate this knowledge that assist in the various activities of programming. We have designed and implemented the CHI knowledge-based programming system, including the "V" wide-spectrum language for expressing both programming knowledge and program specifications. CHI has been used to synthesize several programs including parts of itself. We are extending the uses of the knowledge base to provide intelligent tools for an environment to support not only program synthesis but program acquisition, modification, debugging and maintenance.

Another aspect of our research is called Algorithm Design. This project emphasizes tools to assist in the more creative aspects of the creation of new algorithms. We have formalized a set of methods, primarily focused upon the incorporation of operations into generators, that seem to be a very powerful set of tools in deriving good and difficult algorithms. We have implemented some of these methods in CHI and include a discussion of the derivations in this report.

REPORTS DISTRIBUTION LIST

For ONR Contract No. N00014-79-C-0127, Arpa Order No. 3687 and
ONR Contract No. N00014-80-C-0045, Arpa Order No. 3828

Program Management/MIS
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209 — 3 copies

Commander Ronald Ohlander
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209 — 1 copy

Mr. Marvin Denicoff
Information Systems Program (Code 437)
Mathematical and Information Sciences Division
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217 — 2 copies

Defense Documentation Center
Cameron Station
Alexandria, VA 22314 — 12 copies

Code 200
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217 — 1 copy

Code 455
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217 — 1 copy

Code 458
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217 — 1 copy

Accession For
NTIS GRA&I ☒
DTIC TAB ☐
Unannounced ☐
Justification per
Ltr on file
By
Distribution/
Availability Codes

Dist	Avail and/or Special
A	

Branch Office, Boston
Office of Naval Research
Building 114, Section D
666 Summer Street
Boston, MA 02210 — 1 copy

Branch Office, Chicago
Office of Naval Research
536 South Clark Street
Chicago, IL 60605 — 1 copy

Branch Office, Pasadena
Office of Naval Research
1030 East Green Street
Pasadena, CA 91106 — 1 copy

Technical Information Division (Code 2627) Naval Research Laboratory Washington, DC 20375	6 copies
Dr. A. L. Slafkosky, Scientific Advisor Commandant of the Marine Corps (Code RD-1) Washington, DC 20380	1 copy
Advanced Software Technology Division (Code 5200) Naval Ocean Systems Center San Diego, CA 92152	1 copy
Mr. E. H. Gleissner Computation and Mathematics Department Naval Ship Research and Development Center Bethesda, MD 20084	1 copy
Captain Grace M. Hopper (008) Naval Data Automation Command Building 166 Washington Navy Yard Washington, DC 20374	1 copy
Chief C^3 Division Development Center MCDEC Quantico, VA 22134	1 copy

Abstract

This report describes research in knowledge-based programming and algorithm design during the period 1978 to 1981. The report includes a brief discussion of the PSI system that was completed in 1979, and provides a guide to the literature about PSI.

The emphasis of the report is on more recent research on the CHI knowledge-based programming system and on the closely related Algorithm Design project. Documentation on several aspects of these projects is available for the first time in this report, and much more documentation will be forthcoming during the next contract period.

The object of our research is the codification of programming knowledge and the creation of computer systems that incorporate this knowledge that assist in the various activities of programming. We have designed and implemented the CHI knowledge-based programming system, including the "V" wide-spectrum language for expressing both programming knowledge and program *specifications. CHI has been used to* synthesize several programs including parts of itself. We are extending the uses of the knowledge base to provide intelligent tools for an environment to support not only program synthesis but program acquisition, modification, debugging and maintenance.

Another aspect of our research is called Algorithm Design. This project emphasizes tools to assist in the more creative aspects of the creation of new algorithms. We have formalized a set of methods, primarily focused upon the incorporation of operations into generators, that seem to be a very powerful set of tools in deriving good and difficult algorithms. We have implemented some of these methods in CHI and include a discussion of the derivations in this report.

Contents

Section I

Introduction

This report covers the period from 27 November 1978 to 31 August 1981. This report is a combined Final Report for ONR Contract No. N00014-79-C-0127, Arpa Order No. 3687 and ONR Contract No. N00014-80-C-0045, Arpa Order No. 3828.

Research in the first part of this period includes the latter portions of the PSI program synthesis projects. The PSI project ran from 1974 to 1979. The PSI project is covered in numerous papers, books, and dissertations. Section 3 points out important publications on PSI that appeared during this contract period, and section 10 contains a summary of PSI.

This report focuses primarily on more recent research on the CHI Knowledge-Based Programming project and on the closely-related Algorithm Design project. This report makes available in one place several closely related documents on aspects of these two projects. In Section 2 we give a short overview of these two projects. The overview is followed by two reports on CHI, an overview of the design of CHI in section 4, and a discussion of how CHI can be used as a tool for creating expert systems in Section 5.

Results in the Algorithm Design project are summarized in Section 6. Some of the key methods of algorithm design are presented in Section 7. A detailed derivation, using CHI, of an even-squares algorithm is presented in Section 8. A detailed plan for a divide-and-conquer algorithm derivation is given in Section 9.

The CHI project and the Algorithm Design project are now being combined and are still growing. Several documents are in preparation. One is the doctoral dissertation on the CHI design by Jorge Phillips of Stanford University entitled *Self-Described Program Synthesis Environments: An Application of a Theory of Design*. Another is an elaborate discussion of algorithm design methods by Steve Tappel. We anticipate that some algorithm design results will also appear in Steve Tappel's doctoral dissertation. Beverly Kedzierski, of the University of Southwestern Louisiana, is preparing a forthcoming doctoral dissertation, *Codification of Communication Knowledge for Extending Evolutionary System Environments*. Stephen Westfold is researching a doctoral thesis at Stanford University exploring the power of self-application in programming environments. These research projects are being continued at the Kestrel Institute, 1801 Page Mill Road, Palo Alto, California, 94304. Requests for additional information should be sent there.

Section II

Summary of Progress on Knowledge-Based Programming and Algorithm Design

Cordell Green, Jorge Phillips, Stephen Westfold,
Tom Pressburger, Susan Angebranndt, Beverly Kedzierski,
Bernard Mont-Reynaud, and Daniel Chapiro

Systems Control, Inc.

This section presents an overview of our approach to improving the programming process. We briefly describe an emerging system and set of associated ideas which are being implemented in a knowledge-based program synthesis system, CHI. The object of the research is to codify programming knowledge and use this knowledge to assist in the various activities of programming. A first language, knowledge-base and system has been implemented and is undergoing development. It has been used to synthesize several programs including part of CHI itself. Other ideas focus on the use of the knowledge base to provide intelligent tools for an environment to support not only program synthesis but also program acquisition and modification.

Our approach to improving the programming process is to design an intelligent system to help the user deal with programs. Such a system allows the user to specify programs in a convenient language and then produce efficient implementations. Our previous work in program synthesis emphasized the translation from very-high-level specifications to efficient programs by using the codified knowledge base.

Very-high-level languages do make programs easier to specify, modify and understand, but like all programming languages, they require a supportive programming environment. One way to build a suitable environment is to use a common knowledge-base not only for synthesis but also for specification acquisition, consistency checking, debugging, smart program editing, maintenance, and other programming activities.

A framework for these ideas has been developed by Jorge Phillips. The next subsection discusses our first implementation, based on this framework.

An Implementation - CHI

CHI centers around a very-high-level language, "V", which is used not only to specify programs but to express synthesis rules and meta-rules. V includes as primitives sets, mappings, relations, predicates, enumerations, state transformation sequences, and other constructs. Both declarative and procedural statements are allowed, and facts about program efficiency or algorithm analysis are also expressible. The simple surface form is readable and allows the refinement rules to be compressed in size considerably over our previous rule formalism. A simple compiler for converting high level rules into efficient code preserves execution efficiency. By expressing refinement rules in a clean formalism, their content can be more readily available for scrutiny and transfer to other systems.

The first implementation also includes an object-oriented data base that contains the programming knowledge. In addition to static refinement rules, a dynamic program refinement work space is also contained within this data base and managed by the same set of tools. A data base, or in this case, knowledge-base manager, allows for contexts and multiple versions, and manages file utilities for storing and reloading from disk storage. A structure-based editor is used to modify V programs and synthesis rules. Both user guidance and heuristic search is controlled through an agenda mechanism. The initial knowledge-base has refinement rules for implementing sets, mappings and enumerations using lists, arrays, and hash tables as data structures. Using this knowledge base and set of tools, CHI has been used to synthesize several test programs. The derivations involve fairly straightforward refinement of data structures and control structures.

We use the term *algorithm design* to refer to the more complex and creative aspect of the programming process in which new algorithms are designed, modified and debugged. This part of programming may be contrasted to the more straightforward aspects such as data and control structure selection (though of course a well-defined boundary is difficult to draw). The algorithm design project extends CHI's knowledge-based approach into more difficult areas in order to develop an intelligent set of tools for algorithm design. Thus this research may be viewed as an extension of research on the CHI system.

This research has emphasized methods of incorporating operators into generators to produce significantly better algorithms. An example is an algorithm to find the even squares smaller than a bound. By transforming the even test and the squares test into generators of just the even squares, an $O(\sqrt{n})$ algorithm is produced. To help structure design of more complex algorithms we are expressing algorithm design principles in the form of synthesis plans. The principles include generator incorporation, divide & conquer and store versus recompute. We have analyzed derivations of more complex algorithms including several versions of both "shortest path" (including a dynamic programming algorithm) and "prime finding" (including the Sieve of Eratosthenes and linear time prime finding). This research is described in sections 6,7,8, and 9.

Philosophy & Discussion

Since our research concern is program development, we are primarily interested in programs undergoing change when they are created, modified and maintained. Making changes in a user's target program is often facilitated by making changes in the programming environment. An example is a text editor (part of the environment) which must be extended to edit a new data structure (part of the target program). The tools provided by the programming environment can more easily assist in this modification process if the environment is itself described in terms that the modification tools can deal with.

This idea of using self description has already proven useful. By describing parts of CHI within its own language, V, the system has been used to rewrite and extend parts of itself. In one case, CHI's rule compiler, originally written in LISP, was described in V. CHI then created a new LISP program from the V specification. The high-level rule compiler in V was about 10 times shorter than the original LISP version. It is now easier to include new features, and the code produced is more efficient. The implementation is discussed in [Gr81]. In another use of self-description the knowledge-base manager was extended to include new retrieval commands. Our goal is to extend the self-description to the remainder of CHI.

Is such self-modification really different from that done in other systems? It appears to differ to an extent that may make a difference. Obviously self-referencing is possible in many languages, from machine language up, and bootstrapping is often done with compilers. The notion of a language with an accessible, sophisticated environment expressed in the same language already occurs in SMALLTALK, INTERLISP and others. These systems provided much of the inspiration for our work. But there does appear to be a difference, in that CHI is knowledge-based, and the programs are described in a higher-level description-oriented language. The very-high-level description provides greater potential for the use of the programming knowledge in program compilation and modification.

Another possiblility is that system performance may improve with the addition of knowledge. A drawback of knowledge-based systems is that the addition of new application domain knowledge often slows down system performance. If the new knowledge that is introduced can be utilized by CHI to speed itself up, the speed-up would mitigate the slowdown caused by its introduction. As a simple example, because of the properties of logical AND, the arguments of a conjunction can be expressed as a *set*, and as CHI learns different implementations for sets, it can use them to implement new representation of conjunctive expressions where they are more efficient.

Future Work

A future task is the enhancement of our program editor. The editor is driven from internal descriptions of programming objects. As new programming constructs are added, such as new forms of program synthesis plans for a new application domain, the editor will be able to edit these new constructs and print them in a suitable form. The editor will use an internal constraint mechanism to guarantee that as constructs are created and modified, they satisfy the applicable constraints. This may be thought of as an extension of a modern structure-based editor, in that a structure-based editor enforces the syntactic constraints of a programming language and our editor will enforce more of the semantic and pragmatic constraints of the program being developed or modified.

Since communication is a major problem in large-scale software development and maintenance, another addition is a project management system which will facilitate communication between system builders and the system, as well as users and components of CHI. For example, when a system builder completes a new version of a module, the "description" of the module will be sent to other system builders who will integrate the module, and also to the appropriate parts of CHI. A "model" of the system, along with interaction guidance knowledge, will help in processing user questions and requests and allow bugs, plans, constraints, and other information to be disseminated at the right time. A design of this system is given in Beverly Kedzierski's forthcoming thesis.

References

[Gr81] Cordell Green and Stephen Westfold, "Knowledge-Based Programming Self-Applied", *Machine Intelligence 10*, Ellis Forward and Halsted Press (Wiley), 1981 (forthcoming).

Section III

Discussion of Other Reports and Publications

This section discusses research outside the main area of CHI and Algorithm Design. The first part of this contract period, 1978-1979, included work on the PSI project. An overview of the project, "Results in Knowledge Based Program Synthesis," was presented at the Sixth International Joint Conference on Artificial Intelligence in 1979. This paper is included as section 10 of this report.

The efficiency estimation portion of PSI guides the search for a program by estimating which path will lead to the most efficient program. A description of the efficiency estimation portion of PSI appeared in Elaine Kant's Ph.D. Thesis, "Efficiency Considerations in Program Synthesis, a Knowledge-Based Approach" in 1979. This thesis will appear shortly as a book in the new Xerox series, "Computer Science: Artificial Intelligence," AMI Research Press, 1981. This thesis breaks new ground on automating the efficiency analysis of programs to guide the synthesis process. Her program was able to guide the search for efficiency implemetations of a news retrieval program, several variants of a classification program, and insertion and selection sorts. The process of acquiring time estimates for the coding constructs used in the synthesis phase has been partially automated. A summary of her results appeared in the Sixth IJCAI in 1979, "A Knowledge-Based Approach to Using Efficiency Estimation in Program Synthesis."

Dave Barstow's thesis on the PSI coder was published as a book in 1979 entitled "Knowledge Based Program Construction." It is a part of the Computer Science Library published by Elsevier North Holland.

The dialogue moderator program of PSI chooses appropriate questions for PSI to ask, guides the dialogue, and can answer questions about the topic of discussion. The dialogue moderator is discussed in Louis Steinberg's August 1980 Stanford University thesis, "A Dialogue Moderator for Program Specification Dialogues in the PSI System".

The PSI program model builder was completed as part of Brian McCune's Stanford Ph.D. thesis, "Building Program Models Incrementally from Informal Descriptions," September 1979. This work showed the feasibility of accepting informal partial program descriptions and incrementally integrating them into the program under construction.

The English-language Explanation System produces English documentation of programs. The research focuses on producing coherent, well-structured sentences and paragraphs. These results are described in a Ph.D. disertation by Richard Gabriel entitled, "A Methodology for Mechanical Natural Language Generation with an Application to Program Synthesis," Stanford University, December 1980.

A paper by Cordell Green and Brian McCune on "Application of Knowledge-Based Programming to Signal Understanding" was presented at the Distributed Sensor Nets Workshop, held at Carnegie-Mellon University in December 1978. Another paper by Green and McCune on "Knowledge-Based Programming Applications" was presented at the Workshop on the Application of Artificial Intelligence an Spatial Processing to Radar Signals for Automatic Ship Classification, held in New Orleans in February 1979.

The task of automatically synthesizing a harmonic set formation program fo application to acoustic signal understanding is considered in "Synthesis of a Heuristic Partitioning Algorithm" by Stephen Westfold and Robert Drazovich, SCI Report number SCI.ICS.U.80.2, 1980.

Cordell Green's Ph.D. thesis, "The Application of Theorem Proving to Question Answering System," was selected to be published as a volume in a new series of Outstanding Dissertations in the Computer Sciences by Garland Press in 1980.

Section IV

Towards Self-Described Programming Environments

Jorge Phillips
Cordell Green

Abstract: This section explores the idea of a knowledge-based programming environment based on research in automatic program synthesis and machine intelligence. We argue that the utility of such a programming environment can be enhanced by a self-descriptive capability. The self description allows sophisticated programming tools to be used to extend the environment itself by reprogramming it to incorporate new knowledge. Such extension lets the environment cope with new types of programs being designed. In this section, we present some scenarios illustrating how a self-described programming environment enhances the programming process. We describe the design and initial implementation of one such knowledge-based programming environment called CHI. CHI is based around the "V" language, that expresses both programs and programming knowledge. We discuss how V can be based on an object-oriented knowledge-management system that allows uniform access to all entities described in V. CHI uses constraint mechanisms and a rule-based approach to codifying programming knowledge in order to be very supportive (1) during development and modification of the environment itself and (2) during development, modification and maintenance of target programs.

§1 Introduction

A major practical problem in computer science is how to make computers more useful, both as resilient aids for problem solving and as programmer augmentation tools in which the machine plays a significant role in the programming process. Major research efforts have attempted to provide solutions to this problem. Two important approaches include *programming environments*, where the machine is used as a clever assistant that helps the user manage the complexity of creating and debugging programs, and *program synthesis systems*, where the user is aided in the design and implementation of programs from abstract specifications. In this section, we present a solution in the form of a design for supportive programming environments based on the use of machine codification of programming knowledge, self-description, and program synthesis techniques.

The section is divided into three major parts. First, we discuss how the design of programming environments (PEs) must be centered around the concept of change and how such environments are closely related in nature to other tools for change such as text editors. We exhibit how self-description and system closure become major design considerations to facilitate communication and change. Second, we discuss a design for the CHI environment that incorporates these observations. Finally, we present an implementation for such design and some observations on the use of self-description for self-modification. The CHI self-described environment described herein attempts to provide a solution to the communication and useability problem by being very supportive (1) during development and modification of the environment itself, and (2) during development, modification and maintenance of target programs.

§2 Design of Programming Environments

Our main interest is in the design and implementation of environments to facilitate programming. In this section we study the design of PEs in more detail. The main result of this section is the identification of a need for closure and self-description as a basis for design. We substantiate this result as follows. We start from the observation that applications require creation and change of programs. This entails that a PE, which will support creation and change of programming objects, is in a very definite way quite similar in nature to an editor. We proceed to exhibit why PEs must be designed in a manner in which they can cope with change, and how the process of creating and modifying programs often requires changing the PE. Then we introduce the design concepts of self-description and closure and show how they facilitate change and thus support both use and implementation of a PE. Finally, we show how self-description allows a PE to be used to change itself through self-application of its own programming tools.

To clarify our intent before proceeding, we should mention that in the rest of this section we use the terms *programming* and *programming activity* in their broadest sense: algorithm specification, program design, debugging, coding, maintenance, program management and all other related activities.

2.1 Smart Programming Environments and Editors: An Analogy

An essential aspect of the design and implementation of a programming environment (PE) is the capability of using the environment for its own development and hence for it to be adaptive to change. Change is a basic aspect of any inquiry into the design of programming tools, since most of the time spent developing software is used in changing it. We show here that the use of a PE and the process through which it is designed and implemented may be advantageously viewed as a programming activity. We describe how understanding of the use of PEs may be achieved by 1) viewing them as tools to carry out certain kinds of processes related to the programming activity, and 2) by analyzing the manner in which such processes interact during change.

The closest and perhaps most intuitive example of a programming tool which can be examined within the above framework is the *text editor*. In current programming systems system objects (programs, data, etc.) are created, modified and maintained with the editor. The editor is the main vehicle for change in such systems. What are the dimensions of change that the editor copes with? During its use, a clear cut difference can be observed between change in the program being created or modified and change in the editor itself. Normally the editor is *transparent* to the user which means that the user is unaware of the way the editor achieves what is needed, i.e the user does not need to be aware of the structure or organization of the editor to use it. Thanks to transparency, the user can focus change on the program being developed and work can get done. Most of the time the target program is the subject of change, but there are moments when the editor itself must change. These moments are when transparency of interaction is lost because the editor is not doing what the user desires. In case of such breakdown, extension or modification of the editor becomes necessary. There is a large number of situations when usage of the editor loses transparency. For example, we may consider two situations: one, in which the user is wasting productive time due to a certain editing command not existing, and a second situation where the editor is not performing adequately for the task at hand, for example by not pretty-printing program text being input. In each of these situations editing becomes highly non-transparent: the editor becomes obtrusive. In the first situation, the user would extend the editor by adding a new command in the form of either a macro made up of existing editing commands or a program in some editing language which accomplishes what is needed. In the second case, the user modifies the editor's behavior by telling the editor's printing routine to echo text being input in a nice format that takes into account the syntax of the input language. Use of the editor oscillates constantly between these two states of *transparency* and *breakdowns* (i.e when things don't work). The relevant dimensions of change in the editor are those concerning change in the program being modified, and those concerning modification of the editor itself. An editor will become extremely supportive if its design is geared at being responsive to both changing the target program and changing itself. Only in this manner can the editor aid the user in maintaining a transparent interaction with it. Examples of successful efforts in system design that acknowledge the above observations are the editor EMACS (Stallman, 1979), and the programming system INTERLISP (Teitelman, 1978).

A finer division of modes of change can be observed within the two dimensions of change mentioned above (change in the tool and in the product) by analyzing the ways in which transparency is lost during interaction with the tool. There is a break in transparent interaction with a tool when something needs to be done with the tool and the user doesn't know how to achieve it; when some unintended action is performed and becomes obtrusive to further work; when a new tool that is not at hand is needed; or when a tool behaves in an unexpected way. Design of the tool should take these contingencies into account and ease their repair. For example, an editor should be supportive when a typing error is introduced, a paragraph dissappears, or a macro needs to be written for carrying out some complex transformation. Similarly, in the case of a programming environment there should be tools like structured-oriented editors, cross-reference aids, debuggers, etc. which provide support in those situations where non-transparency arises during programming. These dimensions and modalities of change are applicable to all tools and reflect a natural decomposition of their usage.

We may summarize the above observations in three main points. First, the use of a tool oscillates between transparency and repair modes. Second, there are definite ways in which tool usage loses transparency. Third, the power of a tool resides in its capability for maintaining and reverting to transparent interaction by coping with these modes of change. Thus, it can be concluded that central to the design of a programming tool is the design of a language for communication with the user. The power of the tool corresponds to the power of the language in aiding usage of the tool to return to transparency, and the power of such a language lies in its *expressibility*, i.e. how easy it is to say in the language what needs to be said, and its *extensibility*, or the ease with which the language can adapt to the environment in which it is used. more time in creative efforts.

2.2 Programming Environments Deal With Change

By analogy with the discussion on the text editor as a programming tool, it is possible to identify two clear cut modalities of change in a PE: change as applied to the system itself, and change as applied to program objects being developed (target programs, data bases, etc.) At one extreme creation, repair and maintenance of the system, and at the other, creation, repair and maintenance of generated programs. These two dimensions are unifiable within a single framework as is shown in the next paragraphs.

Let us view both the processes of building and using a PE as programming, and consider the implications. The process of building the framework for a PE is essentially the programming process. Thus, the process of building a PE is not any different from the process of using it, since both usage and building are merely programming. If a PE is to cope with change, it should better cope with both change in itself and change to the target product. This view is shown below:

$$\boxed{\textbf{System Using} \equiv \textbf{System Building} \equiv \textbf{Programming}}$$

This has serious implications on a theory of design and implementation of such systems for then insight into how to build them may be obtained from insight into how they are going to be used.

2.3 Self-Description Facilitates Change

In previous sections we have suggested that frameworks for PEs should be supportive of hange and suitable for the repair of breaks in the transparency of interaction. We introduce two new related design concepts to achieve such behavior: *uniform referenceability* of system components, and *closure* of the PE with respect to its own structural description. Uniform referenceability is used to denote the capability of any system component to access any other component in a uniform system-wide manner. Such access requires a structural description of the system that is accessible to all components. Uniform access to the description, which would contain the necessary information about any system object, would provide homogeneous access between parts of the system. Such descriptions and access primitives are part of the communication language to be designed. We will use the term Closure to denote a property which describes a system that can access, maintain and modify its own description. This concept is thus closely related to uniform referenceability. It differs from it in that closure implies that the system description is part of the uniform reference scheme, i.e. composed of system objects and capable of being referenced and modified by any other object.

The building of a system to support machine-aided programming can be simplified if the system can be used to help develop itself. Self-description, closure and uniform referenceability contribute to this by allowing the system to be used for its own modification, guided by its interaction with the user and its own structure (rules, objects, etc). They facilitate the writing of rules, constraints and metarules that capture strategies for repair, adaptation, and consistency maintenance, since the rules and constraints can be expressed in general terms and interpreted in terms of uniform description and access primitives. This will be the case for a system like CHI.

Closure is a very powerful system design concept. A form of it has been used in interactive programming environments like LISP systems, and has contributed to their utility and success. It is important to clarify in what way the closure we are proposing is different from say, that of LISP. In LISP, the programming tools can be used to operate on each other: the editor, compiler and interpreter reference, use and modify each other. Nevertheless, the access is not uniform. The editor and compiler use very different access primitives to system objects. Moreover, there is no structural description of the system accessible to the system itself. For example, the editor has no way of telling whether a certain modification to the compiler may have adverse side-effects on other components of the system. In a certain sense, a uniformity of reference is achieved thanks to the syntax-directed semantics of LISP. We use the term *surface closure* to denote this kind of description-impoverished closure. The closure we propose for a PSE is a natural extension of the surface closure of INTERLISP, where the system is closed with respect to its description. In other words, the system may be conceived as consisting of a set of components which are instances of general descriptions of the objects needed in program synthesis (which are nothing but programming objects), and which may access either their descriptions or other object instances.

In the design and implementation of a framework for CHI, the vehicle for achieving descriptive closure is the language used for communication with the system. Such language doubles as a description language in which system objects and behavior constraints can be expressed. This language allows a wide spectrum of possibilities for description, from declarative descriptions (like rules), to procedural descriptions (like very high level programs) while allowing arbitrary level of intermixing of these. Thus, the language used for describing programming and program synthesis objects is the same as that used for programming. In other words, there is no difference in communicating with the system to modify it or to use it. Here lies the comparative advantage of descriptive closure versus surface closure.

In summary, descriptive closure is desirable in a system designed to cope with change and to provide transparency of interaction. This kind of closure has two associated system properties: uniform referenceability and self-description. Any framework for a PSE must exhibit these properties. In other sections we discuss how closure can be used to an advantage in order to support system-user interaction when the interaction needs transparency, e.g. when the user doesn't know how to proceed next, or when the user has led the system's state or the program being developed to a state inconsistent with the system's description.

§3 Some Scenarios

This section illustrates some of the capabilities we envision that a self-described programming environment may have in the future. Before proceeding with the scenario we will attempt to summarize the salient features of the design to help the reader understand it. A key point to remember is that the scenario we will exhibit shows CHI being used to support not only its usage but its own modification. It will show how CHI allows self-description and the ways in which the system is supportive of change. The reader should be warned that the different subsections that follow are to be interpreted as desiderata for the behavior of a future system of this nature. Communication with the system will be shown in an informal language for purposes of illustration.

3.1 CHI as a Programming Knowledge Base Manager

The simplest kind of communication a CHI user can have with the system is access to the data management facilities. These aid the user in finding objects that satisfy certain characteristics, forming sets of objects for further consideration, etc. The communication language serves the role here of a query language about the object base.

Typical user requests might be:

▸ What is the type of elements in the domain of mapping M?

▸ Which transformation rules take relations into sets?

- Edit all procedures in CHI's rule compiler that reference sets of integers.
- Find all rules that satisfy predicate P and change them as follows: ...
- Find all sequences of rule applications that can take relations into distributed representations.

Some of these will be given later in CHI's V language. As an example of what the language looks like, consider the editing request. It would be phrased as

$$\text{edit}\{X : \text{procedure} \mid X \in \text{RuleCompiler.body} \wedge \exists Y.\, Y \in X.vars \wedge Y : \text{`implicit set'}\}$$

The above expression tells the system to make the current editing focus to be the set of procedures in the system's rule compiler which have implicit set variables defined in them. The editor will edit each of these procedures. Notice the use of attributes, class constraints and object descriptors like *implicit set*. In later sections we go into more detail on the nature of this language.

3.2 CHI as Library Manager

Programs can be described as objects themselves, using the same kinds of mechanisms used for object description. Programs can thus be classified and accessed from program repositories, in terms of attributes given by the user and accessible to the system. Some of these attributes may be explicit (i.e. stored as part of the program object), some of them may be implicit (i.e. computed by analyzing program objects). If programs are treated as objects in CHI, it is clear that with the organization the system has, a user may define certain program classes as object descriptions that capture the attribute sets relevant to the class. Since programs accessible in libraries to the system would be instances of descriptions in the system, then the same data management interface can be used for accessing program bases as for accessing any other programming object. The system can aid or augment the user in the classification process by verifying for the user that a program has certain particular attributes that justify making it a member of some program class. In general, an inference capability is required to find programs having combinations of desired properties.

One example of this capability is,

> "Find all program sketches in the combinatorics library that do any kind of tree search, and all schemas for doing clustering."

Another example is

> "Find a program that searches graph G for a node N. It may use any graph traversal algorithm."

3.3 CHI Support During Breakdowns

Here we show several ways in which CHI could support the user when transparency of use is lost using constraint and consistency mechanisms. Editing is the main activity through which a user creates objects. CHI is supportive during editing by preserving internal system consistency after user accesses. For example, assume we are defining a variable we previously postponed defining (which is allowed since the system accepts informality in specifications). The following is a possible account of the interaction.

> "The user declares variable X to be a set of pairs. CHI informs the user there is an inconsistency by reasoning as follows: the program dynamically adds components to elements of X in procedure P, this implies that X is a set of objects that grow, but pairs can't grow. CHI then exhibits the relevant sections of the program where X is referenced in the above manner, including additional relevant information to display. The user changes the variable X to be a set of sets of pairs."

The above example shows how the system object-reader is context-dependent, and helps the user to develop a consistent specification by notifying the user of constraint violations. Along the same vein, at any stage in the development of a system or program the user can interrogate CHI about the consistency and completion state of the specification with respect to the set of constraints active at any moment. A possible interaction where this shows up is given below. We will use an English dialogue format in some of the examples, but the reader should keep in mind that in CHI a more formal language (called V and discussed later) will be used instead of English. Also, we do not envision that each English sentence corresponds directly to a V construct.

> *User modifies the rule compiler*
>
> **User:** What type definitions are missing?
>
> **CHI:** The type definition for "AGENDA" is missing and cannot be inferred.
>
> **User:** What else is missing?
>
> **CHI:** The initial value of variable Y in procedure PARSE1 is missing. There are no steps for the inner loop of function F3.
>
> **User:** Initialize that Y to be GetNextConjunct(X Z).
>
> **CHI:** Done. The first step of PARSE1 is now the initialization of Y.

This kind of support during specification is extremely useful for managing the complexity of development of large pieces of software. Some programming environments acknowledge this observation. For example, SCOPE (Masinter, 1980) provides some of the kinds of support described above.

The following example shows how a system of the nature we are describing can be supportive in the moments of breakdown and provide suggestions to the user for the possible directions to take during repair. Assume CHI has an agenda, which is a list of goals to be achieved, and that transform rules place goal requests in the agenda by adding them to the list. We will show how a user could change the structure of the agenda into a partial ordering of goals, ordered by dependencies, by changing the agenda and the rule compiler using support from CHI.

- CHI informs the user that the agenda is currently structured as a list of goals.
- The user redefines the agenda to be a partial ordering on goals, where the active goals are the greatest lower bounds of the partial order. Goals are to be added to the agenda by a new operator on goals called ENTER-GOAL, which is defined elsewhere.
- The user then asks what else should be modified.
- By analyzing the appropriate constraints CHI reports that all transform rules must be changed since they post goals in the agenda.
- The user requests that the rule compiler change all transformation rules that operate on transformation rules by having them add goals using ENTER-GOAL.
- The system modifies the rule compiler's transformation rules. It then uses the rule compiler to compile all rules again, in particular the rule compiler rules.
- Then all transform rules are recompiled. Thus, the system bootstraps the changes into all rules in the system. The system is now using the new priority mechanism.

Much more detail would be required, but the above example suggests a complex intertwining of self-referentiality, closure and constraint management. A system which provides all these for software development can achieve a very close coupling with the user and be extremely responsive to change. A more concrete example follows where constraints are used on the description base. The following constraints reflect conditions that the description base must satisfy to be considered to be in a consistent state with respect to the rest of the system.

- All descriptions are either direct instances or specializations of instances of the description description. No description can violate this rule.
- For any description there is a mapping from the name of the class it describes to the description itself which must be defined if the description exists.

The following example shows repair and support when these constraints are violated during runtime of CHI.

User: I want to edit all sets used in the CHI editor.

CHI: Error: Can't find the description for set!

User: Show me all constraints on descriptions and names

CHI: 1) There must be no dangling descriptions i.e. that are not in some specialization chain of the description description. 2) The mapping M1 from names to description must be defined for any existent description.

User: Verify these constraints.

CHI: Constraint 2 has failed. there is no map value for **set** under M1.

User: Ok. Is there any description called **set** in some specialization chain?

CHI: Yes. Its address is #154236

User: Ok. Define the map M1 from **set** to be this description. Whenever you detect a similar circumstance in the future, patch the system in this fashion, previously reporting to me that you are going to do so.

3.4 CHI as Supportive System Development Tool

We will now assume that the user has described the lower base of the system to CHI and is going to use CHI to modify the lower base to satisfy some requirements, such as efficiency in execution. For the purposes of the following example, CHI will be viewed as follows. CHI operates on a collection of abstract objects. These abstract objects are defined in terms of their structural parts, and predicates that distinguish them from some background. An object can then be viewed as a set of attribute-value associations. As discussed in the section on object management, some objects serve as patterns or descriptions to create other objects. The lower base of CHI is a context layering mechanism on objects and object descriptions. All programming support tools are built on top of this base. Using this short description we will show how the user can use CHI to modify a part of CHI. Consider CHI's rule compiler that translates high level transformation rules into efficient LISP code. The rule compilation process is itself just a transformation of a particular V program (the refinement rules and meta-rules that constitute the compiler), into LISP. If new program synthesis knowledge such as knowledge about hash tables or bitmaps as a better way to implement sets in certain cases were entered, then CHI could recompile (i.e., rewrite) itself and run faster by using hash tables and bit maps where appropriate. This self-improvement capability is made possible in theory by our system design, but it is not clear how soon such self-rewriting capability will become a reality. The following excerpt illustrates a possible view of the process.

The user is using CHI to do some programming, but the system is becoming increasingly obtrusive due to lack of storage space. The user decides to repair CHI...

User: Where is most space currently going into?

CHI: The representation of objects.

User: Objects are implemented as lists of attribute-value pairs.

User: What percentage of such values is boolean?

CHI: 30%.

User: Find the ways boolean-valued attributes may be represented.

CHI: Attribute-value pairs may be represented currently as property lists.

User: Add a new synthesis rule that takes boolean attribute-value pairs into boolean mappings, and a rule that takes these into bitmaps.

CHI: Ok.

Now CHI represents boolean-valued attribute value pairs as bitmaps where appropriate.

User: Transform all boolean attributes in all objects in CHI to be bitmaps using that rule. Store the bitmap representation as part of the object.

CHI: Ok.

Here the user recompiles CHI using CHI, and the system's utilization of space decreases. The system thus runs more efficiently.

The previous example gives some insight into the power of self-description in PEs. Supportive communication and interaction may be achieved with systems like CHI that have access to the relations between their constituent components and to the description of these components. We envision CHI to ultimately converge to a state where CHI is a compiled version of its description in the V language.

We proceed now to discuss how the general approach outlined in the introductory paragraphs may be used to achieve a concrete design for CHI which exhibits the desired features discussed in this section.

§4 Towards a Design for CHI

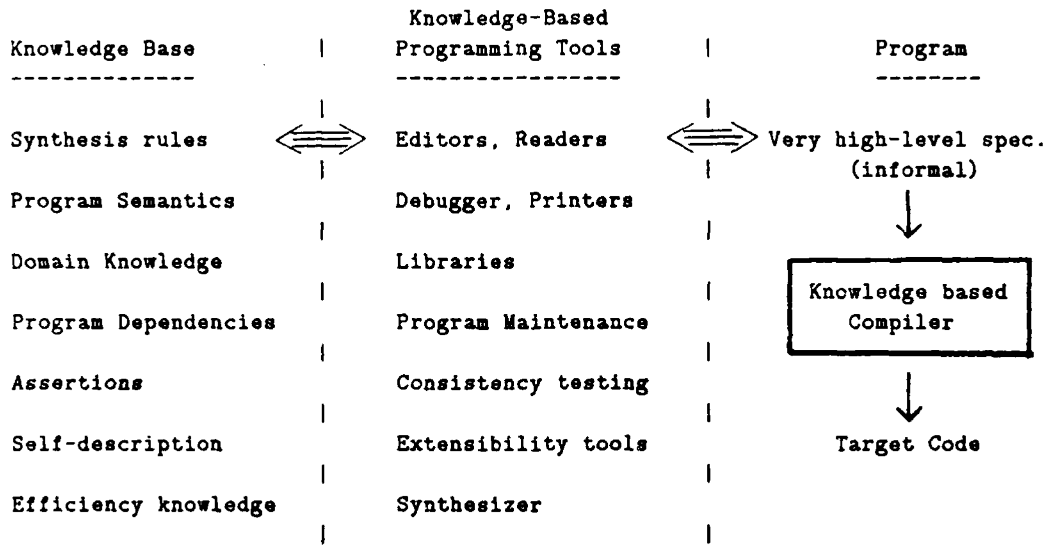
The preceding sections have discussed the foundation for a theory of design of self-described supportive PEs. They have discussed how the nature of the coupling of man to machine is an important theme in the design of systems to aid in the programming task, and how closure, uniform referenceability and self-description facilitate supportive interaction. The discussion of change and use of PEs has produced two main results: a common language to describe both programming of target programs and changing (programming) the environment itself, and the characterization of the kinds of change PEs must deal with. Our approach to PE design consists of four steps: identification of the basic concepts needed for the description of programming processes (i.e. both change to a program and to the system), definition of a language to express these basic programming concepts, design of a framework that allows self-description and embeds the language, and definition of a knowledge base that helps automate portions of the programming process. This section discusses in more detail steps towards a design for CHI, a pilot self-described PE.

The emphasis in the design for CHI is on supportive interaction. This will allow system designers to better understand and improve the environment and to focus efforts towards building a practical repository of programming knowledge and tools. Either the designer or the user should be capable of filling in details in the environment's programming processes that can't be provided by the environment's knowledge base. There are four major points that the design should fulfill. First, the system to be built must include and use a large body of programming knowledge. Second, this programming knowledge should be useable by CHI or the user in the task of programming and adding knowledge to CHI. Third, knowledge should be accessible by function, i.e. by content, rather than by location (referent or name). Finally, the system must be oriented towards use as a PE for experimentation and research on program construction and algorithm development.

The structure of this section is divided into four parts. First, we show alternate perspectives on CHI to give insight into the concrete solutions we propose. Second, we present an initial design for the system. Third, we discuss the lower base of the system (the Object Management base) in detail as the main vehicle for closure. Finally, we introduce the V programming environment language.

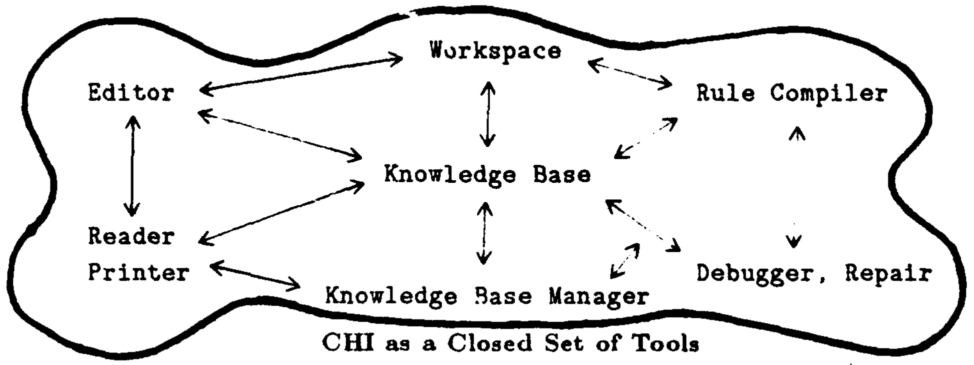
4.1 Perspectives on CHI

Different perspectives on CHI provide different kinds of insight into the characteristics and constraints that the design must have and satisfy. CHI may be viewed at very different levels ranging from that of a compiler to that of an environment. Lets pursue this further. At one level we may view CHI as an extremely smart compiler which transforms program specifications in the V language into executable code. At a higher level, CHI may be thought of as a set of tools that manipulate all components of the compilation process, from the specification to the target code, including the smart compiler. At another level, CHI is a set of rules about how to carry out the translation process, how to use tools, and how to apply the rules to these tasks and perhaps even to themselves considered as part of the programming process! Yet at another level, viewing CHI as a closed system in which all components can access each other, CHI is all of these at the same time. The rules, the tools and the smart compiler can all operate on each other. Thus, for example, the compiler can compile rules that tell the compiler how to compile rules, and the compiler itself may be just a set of rules, so it can recompile itself, etc. The number of possible uses of tools thus becomes very large in CHI. This last view of CHI as a closed system provides an important perspective on the design for CHI as a knowledge-based PE, i.e. a programming environment oriented not only to programming of programs, but to the programming of programming knowledge bases, i.e. rule bases and structures that aid in prog.amming. It may be considered then as a tool for the development of knowledge based systems. The following figure suggests how programming knowledge and tools bear on the smart compilation process under this view.



CHI as a Knowledge-based Programming Environment

The above figure should not mislead the reader. Although processing in CHI may be viewed hierarchically for simplicity, it is really distributed and at a single level. We have introduced the hierarchy above as a device for showing a particular view of CHI. The reader should realize that closure places all process elements at the same level. A better view then is to consider CHI as a conglomerate of tools, programming knowledge and a description of itself. These and the programs under development constitute a workspace which is a natural extension of the knowledge base and which is implemented on a uniform representation that allows environment tools to modify themselves and to be applied to other components including the environment's description. The following diagram exhibits this other view. Arrows in the figure reflect these uniform access and applicability relations.



CHI as a Closed Set of Tools

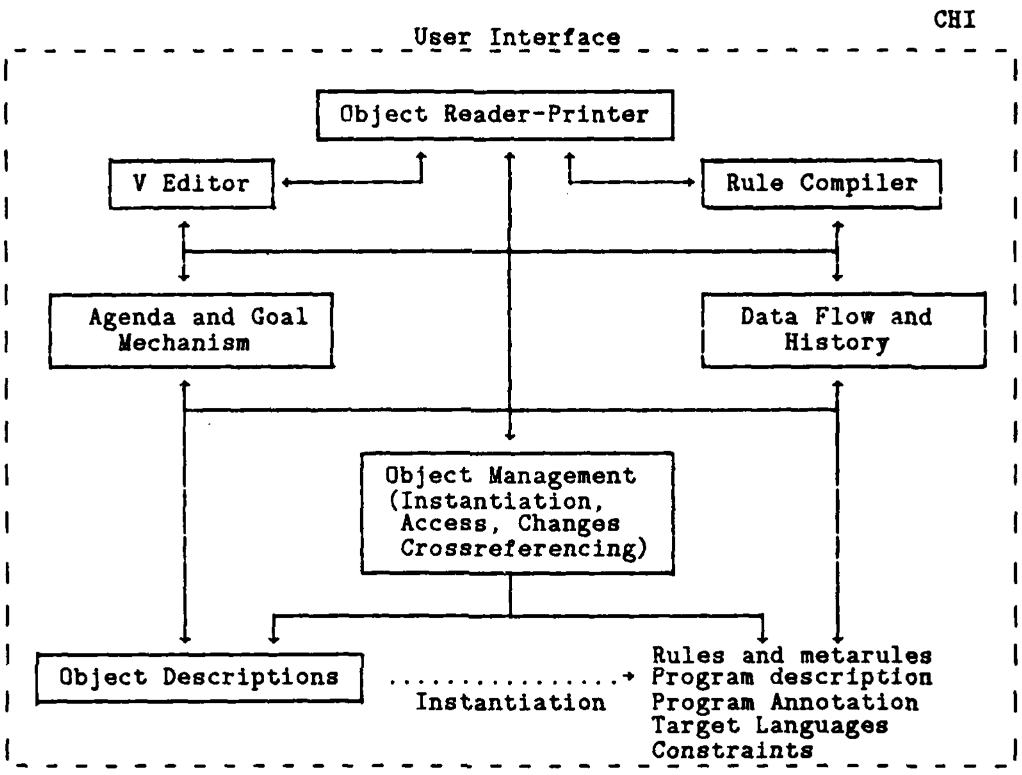
4.2 An Initial Design

The initial design for CHI must provide a foundation on which to base the system's further development. The foundation requires identification of a set of primitive system usage and building activities, and of an initial set of programming objects to be manipulated. Once these are identified it is possible to proceed with the design of a conglomerate of tools that can be used to carry out these activities on the programming objects.

The most frequent activity in a PE, and that in which users spend most of their time is in the creation and modification of programming objects using the editor. This is no coincidence, since the editor is the locus for object generation. It is mostly during *editing* where the system is changed by the community that uses it. CHI must provide a smart editor to the user. As a matter of fact, it would be highly desirable if any object dealt with by CHI (including objects related to the process of running the system itself, like node transformation sequence objects mentioned later) could be dealt from the editor. We contend that systems of the future will look basically like extremely powerful editors to their users. A user of such systems will always operate within the editor. We may expect that in CHI the editor must be the most supportive tool since it is where most breaks of transparency will occur. Another activity essential for transparency of the system is *analysis*. When engaged in this activity, the user finds out properties either about the system itself (i.e. about objects that constitute the system), or about the program being developed. Support for this activity means having a smart program management facility. This facility must provide data flow information, internal indexing and cross-referencing, and similar information. *Consistency Maintenance*, somewhat related to analysis, is used to enforce and verify that the system satisfies its design constraints, i.e. that it doesn't violate its description. Usually consistency requirements are specified as invariant relations that must be maintained between the system's implementation and its description. Another aspect of analysis which we are exploring are facilities for *process examination*, where the user may interrogate the system about the system's behavior and history. Finally, it must be possible to execute the system. This means that the system must provide a facility for translating descriptions of system objects to some substrate where they can be executed (by substrate meaning some host environment, like in INTERLISP in the case of CHI). In other words, CHI may be viewed as an extension of INTERLISP. In CHI we may expect to have objects maintained in several different representations, some suitable for manipulation, analysis and examination by other system components, and others suitable for execution.

Now that we have defined the main activities that must be supported in the initial design, we may look into what needs to be manipulated, i.e. which programming objects are needed. Since we will talk about objects in the system later, in our discussions of the V language, we will only mention briefly a few so the reader can get an idea of the spectrum of objects we are thinking about. Objects in CHI include common very-high level programming objects (V constructs), rules, metarules, constraints, objects related to the history of execution of CHI, program annotations, and target language constructs, to mention just a few.

The activities we have mentioned already imply having a specific set of tools. The basic tools available in the initial system are a smart editor, an object reader and printer, rule compiler, and machinery for synthesis and analysis (in the form of process agendas, goal mechanisms, planning components, data flow analysis and system behavior examination tools). All these operate on the object management component of CHI (where objects are created, modified, etc.) which provides a uniform descriptive interface to all the components of the system. The initial organization of CHI is shown in the following figure. The reader should recall again that there is no implication of an internal hierarchical organization of the system in the figure. Since all system tools are to be objects themselves, an accurate depiction of the system's organization would be one in which all tools and objects are part of the object base. Such a figure is given in later sections when we discuss the relationship between closure and CHI's design.



The CHI System: Initial Design

4.3 The Object Management Base

In the preceding paragraphs we have mentioned briefly the object management component of CHI. This section discusses it in more detail. Here we attempt to give an account of *the way in which we view objects*, and some internal details of the implementation of an object base for a self-described PE, and in particular for CHI.

A central aspect of a PE, as discussed previously, is its use as support tool for manipulations of programming objects. A user of a PE is constantly engaged in the creation and modification of programming objects and of purposeful interconnections of these to achieve desired behaviors. Such a user develops programs and specifications through processes that involve *creation*, *destruction* and *modification* of programming objects. These processes are the same be it for system or target program development. The data management portion of CHI provides a repertoire of programming objects and operations that can be carried out on them. CHI is then a mediator of object manipulations expressed in the V language discussed later.

$$\text{Manipulations} \xrightarrow[\text{The V Language}]{\text{CHI}} \textit{Objects}$$

CHI as Mediator of Object Manipulations

The system is intended to be used as a design and implementation tool, and thus has as main component a data management module for operations on program objects. This section discusses the nature of such objects and of the operations performed on them, and the role they play in the design of CHI.

4.3.1 What is an Object?

In the preceding discussion we have been mentioning objects. But what are these objects? We will consider an object to be a collection of distinctions performed by some observer (user or system designer, which may be the same person) against some background. Distinctions are made in terms of attributes and their associated values. Attributes take the form of predicates that hold for the object, or values of applications of functions to the object. Notice that attributes are objects themselves and can be arbitrarily complex, their values including functions, variables or predicates.

We view the role of objects in a PE to be threefold: they are a mechanism for self-description, an internal representation for the process language used by the PE, and a mechanism for storing and retrieving attributes about the process of programming and programs in use or under development. The above definition of objects is quite similar to the method for description espoused in KRL-like languages (Bobrow & Winograd, 1977), but is viewed under a different orientation. As an example, the following are object descriptions in CHI:

```
                A loop with no exit test

      The set of functions that return boolean values
```

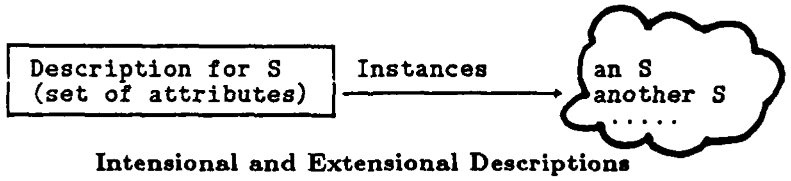
Objects in CHI

Objects may be implemented by any suitable representation of sets of attributes. This representation may be *explicit* if objects have storage assigned to them, or *implicit* otherwise. In the first case, objects are stored and can be considered as the internal implementation of some abstract syntax for an interaction language. In the second case, objects are computed and can be viewed as the result value of access operations to the object data base.

4.3.2 Object Classes

The object base in CHI holds all objects the system can manipulate, including objects that are part of CHI. These objects correspond to common recurrent concepts found in the design and implementation of programs, and in the carrying out of synthesis processes. They include very-high level programming constructs, attributes that may be used to define or modify objects, synthesis and search control objects such as agendas, tasks, rules, metalevel rules, and constraints on system operation and system structure, and also operations and actions that may be carried on these system components.

These recurrent concepts correspond in a well-defined manner to recurrences in the programming domain, i.e. concepts that become manifest over and over in the activity of programming. An important concept in the development of CHI's object base is the idea of *class*. Intuitively a class corresponds to a set of objects which share the same pattern of attributes. If we have a set of functions and predicates that can be used for defining patterns, then patterns become objects (under our definition) defined in terms of distinctions that use these functions and predicates. Using this idea, classes (which we will call *descriptions*), can be made first class citizens in CHI, and be amenable for manipulation using all of CHI's object machinery. Object classes are then *intensional* descriptions. The set of objects distinguished from the background with the set of distinctions in the class description are called the *instances* of the class. These correspond to extensions of the description. In CHI all instances are explicitly stored.



Intensional and Extensional Descriptions

4.3.3 Object Instantiation and Specialization

In the preceding section we have mentioned that in CHI all objects are instances of distinguished system objects called classes or descriptions. The basic operation in CHI is that of constructing an object, which is done by taking a description of the corresponding object class and passing it to the object constructor. We call this operation *object generation*. This operation of construction introduces a relation between objects (i.e. between description and instance of description) that is central to providing inference speed-up for deduction over attributes in the object base, and for expressing the manner in which properties are associated to newly created objects. In the current version of CHI there are two different dimensions of object generation corresponding to whether the new object is a description or not. The first case, i.e. the generation of a description from a description we call *specialization*, and the second case we call *instantiation*. This distinction is accidental and caused by a limitation of the last implementation of CHI that disallows instantiation of objects not explicitly declared to be descriptions. The current implementation eliminates this problem by allowing any object to be viewed as a description. This view is more general since there is only one generation operation, namely producing an object from some other object which is interpreted as a description. What we have called instantiation and specialization would then be special cases of this operation.

```
      DESCRIPTIONS                |              INSTANCES
                                  |
    A Data Object has a Name      
                 |                |
    descriptor   |  specialization |
    inheritance  |                
                 v                |
    A Set is a Data Object and    |
    has a prototypical element - - - - - - - -→ A Set of Records
                                instantiation    called DATABASE
                                  |
```

Instantiation and Specialization in CHI

4.3.4 Closure and the Object Base

Closure poses strong constraints on the object base. As the reader may recall, we have defined closure as the capability of the system for uniformly accessing any of its components or any of the components of programs being developed. Since all manipulation is done on objects, and manipulating an object needs the information present in its associated object class description, all objects in CHI must be instances of some class. A fully closed system, is one in which there are only instances of object descriptions.

Everything in CHI by the *closure* property must be an object, since only in this way do we have uniform access to all of the system from anywhere in the system. As a first approximation CHI is a large collection of such objects expressed in terms of distinctions (i.e. functions and predicates) relevant to the programming domain and programming methodology implicit in the system. Since CHI provides for manipulations, there is a set of operations that may be used towards this end. These operations are objects themselves. Similarly, since it is necessary for the user to be capable of operating on objects directly, i.e. as sets of attributes, thus needing operations on the attributes themselves, then the latter must be objects too. This means that attributes are also objects. In summary, so far we see that 1) Objects are sets of distinctions made in terms of attributes 2) That everything in CHI is an object since the system manipulates only objects, and 3) that anything that is used to achieve these manipulations must be an object itself. Only if these characteristics hold, does the user have a handle on manipulating the manipulation process itself, which is necessary for supportive system design.

If we think more carefully about what this implies, we can list the following six constraints ("Description" in what follows stands for class):

- Descriptions, predicates and functions must be objects.
- All of them must then have descriptions, i.e. have an associated class
- There must be a description for an object, which is the most general (weakest) distinction in CHI. Every other description must be in some chain of specializations of the "object" class.
- There must be a description for a description, which is a description of itself.
- All object operations must have descriptions for them.
- All descriptions are instances of the description description, which in turn is an instance of itself.

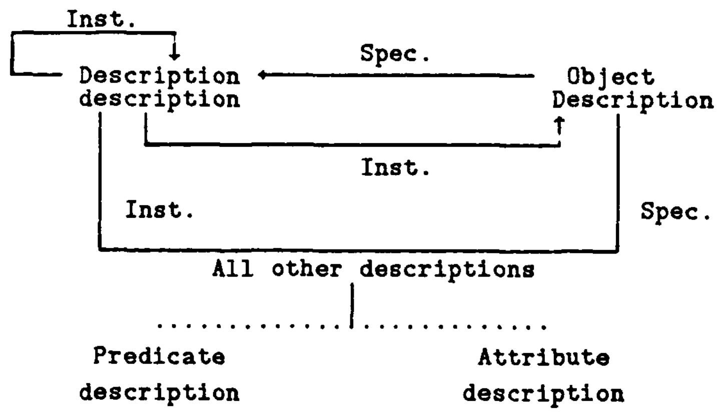
These constraints place definite restrictions on the object management base. We will end this section with a discussion of the organization of the base in CHI.

4.3.5 Organization of the Object Base

We finish the discussion of the object base with a short detour into the role of organization and structure in the base, and a general overview of the way we have set up the class-instance-specialization chains in the last implementation of CHI. In a system we may distinguish between its *organization* and its *structure*. The organization intuitively reflects the set of relations that make the system be a member of its class (for example, what makes a sorting program be a sorting program, i.e. that it takes some input and ordering relation on the input, and produces an ordered or sorted output). The structure on the other hand reflects the concrete relations that satisfy the organization, i.e. how the organization is implemented. A system like CHI, which provides possibilities for design, provides both organizations for programming objects as well as structures that can implement them. As an example of what we mean by organization and structure, consider a *rule object*. The organizational definition of the rule will say that "*A rule consists of a set of pattern variables, an antecedent composed of rule and pattern operators, and a consequent which is a composition of actions*". It also specifies an associated set of relations, constraints and operators that may operate on rules, which we omit here for the sake of clarity. Meanwhile, the structural or implementational part of a rule may say that *A rule may be implemented as a plex (record-like structure), whose fields store implementations for rule components.* Thus, the structure of a rule effectively shows how to carry a rule description into a system generating substrate where the description becomes useable by other system components.

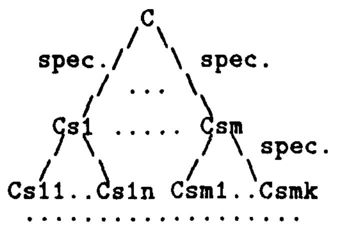
The object base of CHI provides the environment it with a system-accessible system description. This description takes the form of organizations for the environment's possible components (the object classes). It is constituted by a network of class descriptors where arcs in the network denote relations of specialization and instantiation. The system itself consists of a set of concrete components or *instances* of the descriptions, whose implementation or structure has in turn been determined by particular instances in CHI of rules that guide the process of generating structures from descriptions (i.e. a set of implementation rules). In short, the description base provides *organizations* for system objects that describe the constitutive relations of the objects, and *structures* that implement these organizations. The structural component of a program synthesis system thus acts like a bridge between system description and the underlying substrate in which the system is implemented (in our case INTERLISP).

An interesting theoretical problem is how to start the construction of the object description base of a descriptive closure for a certain system. We chose the following organization for the object base that reflects the constraints derived from the closure requirement.



Basic Description Structure in CHI

We now proceed to discuss briefly the meaning of classes and class specializations within the organization we chose for the object base. The idea is that a class denotes a set of objects that satisfy the set of attributes defining membership in the class. These objects include both direct instances of the class description and instances of any specialization of it. The reasoning for this is as follows. In CHI there is a specialization operator (call it SIGMA) which generates new class descriptions. Let SIGMA(C) for a class description C denote the set of class descriptions which are specializations of it. Notice that a specialization of C has instances which satisfy membership attributes in C plus some other attributes which define membership in the specialization. Note that this entails that a class C is by definition a specialization of itself. Then, it follows that a specialization of a specialization of a class C is also a specialization of C. SIGMA(C) thus defines the transitive closure of the specialization chains running from C, and in turn defines the class C as all those objects that are instances of some description in SIGMA(C). All this justifies the fact that in CHI a class C is denoted by a hierarchy of descriptions for itself and its SIGMA transitive closure. Thus,

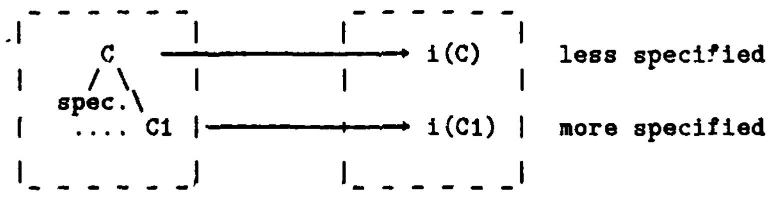


A class C in CHI

In the above figure we have that

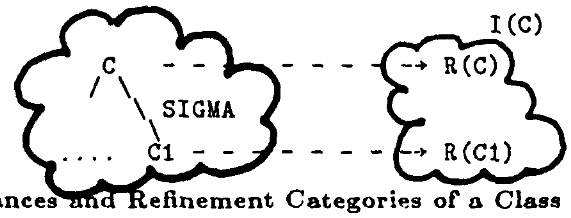
```
Instances(C) = {x| x∈instances(C1) for some C1∈SIGMA(C)}
```

At any level in the class hierarchy the extension (set of instances) of a class at that level is the set of objects in CHI that satisfy the description but do not satisfy descriptions for any specialization extension. From a different vantage point, it corresponds to those objects which have not been specified enough to allow placing them in a more specialized category. For example, currently in CHI a *set* is a specialization of *data* object. If an object is directly an instance of the *data* description, it means that only enough is known about the object to identify it as a *data*. As more and more becomes known about the object it moves along the instances of specialization chains: a *data* may become a *set of integers* when its data type becomes known. We may say then that instances of *data* are in a less refined or specified state than instances of *set*. This interpretation of instances in a specialization hierarchy gives a handle on informality in program specification. Thus, the set i(C) (where i(x) is an abbreviation for instances(x)) denotes a state of refinement. In the following figure objects in i(C) are in a less refined (specified) state than objects in i(C1). Within the class C, this mechanism denotes different levels of detail in specification.



Informality and Descriptions in CHI's Object Base

If we merge the notions of instance sets i(C) with the hierarchy of specializations, the object base in CHI is structured in terms of the concepts of a *refinement level*, i.e. instances at a certain level of specification and *instance set* which is the set of all refinement levels hanging from specializations of a class. We also call the R(C) *refinement categories*.



Instances and Refinement Categories of a Class C

4.4 The V Language

Closure and self-description are desirable elements of a PE. But how do we support closure? How do we manage to give uniform access to all objects in CHI? Answers to these two questions lie in two areas: the use of a uniform descriptive interface for objects (which we have discussed in the previous section) and the use of a single language for system description and use. One of the key observations we make is that a major ingredient in the engineering of self-described PEs is having a language that doubles both as a language in which users can communicate with the system to carry out processes, and as a language which can be used by designers to describe the PE.

Reflecting on the last statement, one arrives to the conclusion that it is logical to require this of the language. After all, both cases involve communication to the system about programs. It is also necessary to do so, since the only means user or designer have for interacting with a PE is some kind of language which they can use to express the interaction. The advantages of having a language of this nature are apparent. If the system is programmed and used via a single language, and if the language is at the same time the descriptive agent in the system, then it becomes possible to use the system for its own development and maintenance as well as for the development and change of target programs. This is not new at all. Systems like INTERLISP have exploited heavily the single language idea: most of the system's code is written in LISP and the INTERLISP programming environment is used for its own development. We propose an extension to the concept of single language. By having a system-accessible description in the single language, that not only describes the language semantics but also the organization of the system, the description can be used as a tool for managing complexity providing both consistency constraints and what could be loosely termed domain support. We will discuss this in more detail in the remainder of this section.

The practical significance of using a single language, and of having a uniform interface to all objects in a PE is that all tools written in the language and operating on the uniform object base can be used on any object in the object base disregarding the role the object plays in the system. In other words, only one smart knowledge acquisition system (editor, printer, program description maintenance, source of semantic support, etc.) need be built for the PE. The implications of this can be seen in several places. For example, a meta-rule can mention other rules in the system; a program specification can reference a previously written program; and for a new input the smart editor can request consistency checks using existing rules and descriptions in the system. The self-referentiality acquired in this manner enables the exciting possibility of a self-improving system.

The design of CHI is centered around one such language called V. This language is intended to provide a general semantics for programming so that it can act as a foundation for transformation of program descriptions into practical programming languages. It also provides capabilities for dealing with descriptions of system processes and descriptions of programming objects. V has been designed to provide the capability of expressing a large fraction of the objects used in the programming process. As such there is a lot of different object families describable in the language itself: programs, rules, metarules, object descriptions, operations, etc. V is also uniformly extensible, i.e. it allows for continuous change and development of the objects used in programming. Only if this requirement is satisfied can the system designer hope for transparency of usage.

Before proceeding any further, the reader should be warned that we don't believe this approach to be a universal panacea. It is evident from practice that the quest for a universal language is unrealistic. Nevertheless, we do hope, despite the fact that size and number of constructs must be maintained at a minimum, that a very-high level language like V may be a unifying framework for diverse families of target languages by being a natural extension of all of them, and may provide a solid foundation for the development of algorithms from abstract specifications. With this caveat we proceed to discuss a particular view of V as a language for programming environments.

§5 A Language for Programming Environments

Let us recapitulate where we stand: we have discussed a theory of design centered around the concept of change, presented a preliminary design for a PE called CHI which embodies such a theory, presented a formalisation of the concept of objects upon which we build a foundation for the manipulation of programming objects, and introduced briefly the benefits of a language called V with which we both develop the system and interact with it. We have called LPEs (Languages for Programming Environments) all languages of the class exemplified by V, to emphasize their uniform and global use in the design, development and usage of self-described PEs. This section discusses the nature of the V language and of its use.

The V language includes as its main procedural component a typed language that allows informality in specification (for example, types may be omitted, variables may be left undeclared, procedures may be only partially defined, etc). The primitive data types of the language are very high level mathematical concepts. The procedural component of the language is used for the definition of component modules of the PE, and for the specification of target programs. The primitive data types in V include sets, mappings, relations, enumerations, partial orders, trees, graphs and other elementary types. The language also includes the corresponding primitive operations for each type. Programs are composed of type definitions, variables, constants as well as procedures and functions. Program structuring is very similar to that of languages like PASCAL or ADA. Other structuring concepts include blocks, value-returning blocks, functional maps, and non-standard structuring operators like producer-consumers and generators.

The presentation of the language in the following paragraphs follows a particular way of viewing the process of using CHI. At its lowermost level CHI provides objects and operations that can be performed on the objects. We may consider this to be the first capability that V must have, namely of expressing how to *change* objects. The next level is introducing logical sequences of operations, i.e. expressing the *order of changes* to objects. The order of changes expresses the relationship in time between changes. At even a higher level, it is necessary to constrain the evolution of the system by constraining the ways in which change sequences may be ordered. For example we may constrain a particular operation on an object never to be executed unless some condition holds on some other objects. V must be capable of expressing these constraints on the *order of sequences* of operations. In summary, the initial concern is *change*, i.e. the capability of communicating with the system to create, destroy or modify objects; the second concern is *order of change* or *process*, i.e. the capability of expressing the order in which the system is changed; finally, the supportiveness of the system is given by the capability of its users to communicate with it about the *order of process*, or order of order of change. This means the capability for designing systems which are adaptive in an evolutionary fashion. Since the user of the system communicates with the system via the language in which interactions occur, it follows that any language we design must cope with this layering of communicative capabilities.

We will choose to present the language in a bottom-up fashion where we build up from a language for talking about programming objects, through a language that allows to talk about change to them, to a language that allows to talk about ways to design order of processing. Therefore, we will structure our presentation of the V LPE into three major subsections dealing each with the use of the language to express change to objects, processes on objects and change on processes on objects.

5.1 Object Change

The previous sections have been talking about the centrality of our description-instance approach to the design of CHI. It should be clear that one of the major considerations in the design of V must be a way for communicating object definitions and descriptions to CHI. In this subsection we show this part of the V language. If you recall from our discussion on the object base, an object is a collection of attributes, a distinguished one of them being the class it belongs to. An object description is a set of such attributes and denotes an object. Object descriptions in V are either logical combinations of attributes or schemas which reflect the external form of such combinations. V provides the concept of a *variable* as a binding from a name to some value object (which may have an associated description). Variables are denoted by their names. An example of object descriptions is

```
X:set ∧ X.element:set ∧ X.element.element:integer ∧ X.ordered
```

or its alternate representation

```
'ordered set of set of integer'
```

Object Descriptions

Both descriptions above define an ordered set whose prototype element is a set of integer values. In the above example we have used two operators included in V that apply to any object. They are denoted by "." and ":". We will use the notation `X.Y` to denote the `Y` property of object `X`, where `X` stands for either a variable or an object description. We will use the notation `X:Y` where `Y` is either an object description or a class name to denote whether `X` satisfies predicate `Y`. Thus for example we may have

`F.domain`	*the domain of F*
`X:set`	*X is a set*
`F.domain.element:set`	*F is a map with domain a set of sets*
`A:'partially ordered agenda'`	*A is an agenda implemented on a poset*
`A:(*.ordered Ξ *.size=3)`	*A is ordered and of size 3*

The . and : operators in V

In the last example we have used the dummy variable * in the definition of the predicate. In V the variable * stands for the current object being considered. The weakest class description in the system is that of `OBJECT` since everything is an object. Thus, by suitably constraining a description of objects we can define any subset of the objects in CHI. In this manner, we can denote the set of all sets in the system by `{X:object| X.class=set}` or by `{X:set}`, which is equivalent since **set** is a specialization of **object** and this means that the set of **sets** is a subset of the set of **objects**. In the preceding examples we have used schemas which are like patterns. an example of a schema is `'partially ordered agenda'`. Schemas are usually enclosed in `'...'`. They are denoted using a pattern like notation which is internally translated to suitable accesses to the object base.

A final remark we must make is that the notation makes no assumption about the attributes. They may be either explicitly stored in the implementation of objects or computed via some method associated with the property itself. This class of notation provides a uniform reference mechanism, and allows the user to dissociate himself from the way the system is implemented. The only thing that is necessary is that the property have an associated description that gives the necessary information to the rest of the system. This greatly aids the task of the system designer by decomposing implementation from denotation. Now, objects are created either as instances which are values of variables (and these may be created either explicitly or as values of operators) or as constants which are like variables whose associated value object is environment-independent. The implicit creation operator is called `create` and it takes as argument a description of what needs to be created.

```
create 'set of set'
create (*:set ∧ *.size<25)
```

Object Creation

Having a way to denote, and create objects, the next logical step in V is to express change on objects. Objects can be changed by adding or modifying properties (asserting predicates, filling in detail in their structure, etc.) The ← operator modifies or adds parts of an object. Thus,

`X.domain ← 'set of tasks'`	*make X's domain a set of tasks*
`X.ordered ← True`	*make X be ordered*

Object Modification

The third example above shows the dichotomy between boolean functions and predicates.

5.2 Processes

Once a means for change has been introduced, the next step in the design of a LPE is how to express the way change occurs. There are several possible ways of doing this. Before this though, we will explore briefly the nature of processes. We can assume a state space approach to the evolution of a system along time. Change is brought about during state transitions caused by the application of operators to objects. These operators are applied either by the system itself or by the user through the user interface. Order of change can then be expressed by either providing explicit procedural orderings or implicit orderings in the form of situation-action rules. In one case we will have programs and in the other rules. In V the procedural definition of changes can be denoted in a manner similar to that of any typed language.

The following simple program exhibits some of the capabilities of the specification language. It shows how type disjunctions (such as the *alt* construct) are expressed and how they are dynamically distinguished. Also, it shows the use of relations and relational expressions in relation inversion.

```
module News
var DataBase : relation
                 Story,
                 Keyword;
    Query    : alt
                 Keyword,
                 Escape;
begin
  input(DataBase);
  loop
    input(Query);
    exitif type?(Query) = Escape;
    Stories ← invertrelation(DataBase,2,Query);
    output(Stories);
  end;
end;
end News;
```

A Relational Query Program

Notice that it is possible to use the object management primitives from within V. This is necessary in order to describe CHI in V. Let us show another example of a V program, in this case a simple agenda.

```
description AGENDA
  parts
    TASKS: partial order on TASK;
  subdescriptions
    TASK = record
             node : object;
             attribute : name
           end;
  operations
    CREATE-TASK (O:object; P:property);
       TASKS ← TASKS ∪ {a TASK with node=O, attribute=N};
    SELECT-TASK;
       select X∈{y|lub(Y,TASKS)},
              R:'refinement rule'
         st applicable(R,X)
         do [apply(R,X!node); TASKS←TASKS-{x}];

  plus
    ...assertions about partial orders...
    ...optional attributes for agendas...
    ...etc...
end AGENDA;
```

A Hypothetical Agenda Described in V

This simple-minded program implements a brute force agenda. Here an agenda is a partial order on tasks, where a task is an attribute to be filled for a certain object. CreateTask inserts a task in the agenda, and DoTask selects some task and applicable rule and applies the rule to complete the task. If so removes the task from the agenda. Notice that attribute access is denoted by . and plex field access by !. These two examples should give the reader a feel for what the procedural part of V looks like.

Another way was mentioned for ordering change based on rules. It may be called the *implicit* or *declarative* way of ordering change. The V language includes the capability for communicating about program construction processes in the form of situation-action rules. The use of this kind of rules for the codification of programming knowledge has been successfully explored in (Barstow, 1979). Currently there are two classes of first-order program construction rules: those that *transform* objects into other objects, and those that attach properties to objects. What is the role of this declarative expression of first-order programming knowledge? It is used for carrying out the basic manipulations on objects discussed above. An example of a program construction rule that works by attaching properties is converting a *projection* of a relation (i.e. choosing a coordinate of the underlying cartesian product and an associated value, and returning the set of relation points that have that value for that coordinate) into an *implicit projection* by adding the property "implicit". An example of a program construction rule that works by transformation is converting a *relation projection* to an *inverse union*, which is the case when the relation and the projected coordinates are represented explicitly by an inversion. Rules as any other element of V can call on any other facility in the system or language, for example, can create or destroy any object in the system.

The following are some examples of typical rules:

1. **A representation of a set:** "A set may be represented by its characteristic function, as a mapping into the set consisting of the elements **true** and **false**". This rules is written in V as

```
        transform 'set of X' → 'mapping from set of X
                                        to set {True, False}'
```

2. **A transformation of a membership test into an existence test:** "A membership test on a stored collection may be refined into a test on whether any item in the collection is equal to the item being tested". This can be denoted by

```
      transform 'x∈S' ∧ S:'stored collection' → 'test ∃y∈S st y=x'
```

Other transformation and program synthesis systems have attempted to provide means for expressing first order knowledge of this sort. We present below a comparison of PSI's rule formalism (Green et al, 1979) to CHI's rule formalism. Consider the rule "*If a collection is represented as a Boolean mapping, a test of whether an item is in the collection may be implemented as a retrieval of the image of the item under the mapping*".

For PSI the rule is expressed as

```
(* Rule GCOLLECTION.4.1 (IS-ELEMENT => GET-CORRESPONDENT))

(DEFINEQ

(GCOLLECTION.4.1
  (REF←[IS-ELEMENT [←P COLLECTION (←← Y)
                     (←RDS (←REF CORRESPONDENCE (←P DOMAIN-ELEMENT (←← DE))
                                (←P RANGE-ELEMENT (←REF PRIMITIVE
                                                      (←P SPECIFIER
                                                          (?←= BOOLEAN]
                   (←P ELEMENT (←← X)
                       (←RDS (?QUERY REPRESENTATION-MATCH ← DE]
    (←NEW GET-CORRESPONDENT (←←P
           CORRESPONDENCE Y)
          (←←P
           DOMAIN-ELEMENT X)))))
```

while for CHI it is expressed as

```
      'X∈S' ∧ S.TRANSFORM:'BOOLEAN MAPPING' => 'S(X) = TRUE'
```

5.3 The Control of The Evolution of Processes

So far we have seen one possible manner in which a LPE (in this case V) can be used for introducing change into the object base, and for expressing order of change. At least as important is the capability for a designer to be able to cope with the recurrence of breakdowns of the system and to provide a structure that guides the evolution of the system. We don't claim here that it is possible to predict the space of possibilities for a system, but that it is possible to design the breakdowns the system will enter in. In short, it is necessary to be capable of avoiding the occurrence of breakdowns, and of guiding in this way the space of processes of a system. This is perhaps where the crux of the design activity lies. In this section we study the mechanisms available in V and thus in CHI for management of breakdowns, supportiveness, and transparency.

What happens when we use CHI? Here we rely partly on the understanding of the reader of the nature of the process of programming, and partly on the light shed by the use of CHI by the authors. Recall that CHI is built on top of an object base, and that we have operators and facilities for activating operators procedurally or in a declarative manner through rules. In what way can we fix the state path of the system to lie in a certain subspace determined by the states of individual components? The basic operation here is *constraining*, which can be done in several ways. Two possible ways are *explicit* constraining whereby state changes are triggered by situation-action rules, or *implicit* constraining where the constraints are triggered when a certain state is reached. Thus if in one case we have that there are components that are determining the next state or space of possible next states, there are other passive components that serve as detectors where repair for the upcoming breakdown may be attached. Again, some of them force states, some of them redirect states. We believe that these two are equally important in design. The rest of this section is spent discussing the entities found in design breaks in programming systems, and on the means for constraining their evolution during system usage.

5.3.1 Programming Process Entities

The behavior of a system for systematic program construction like CHI can be characterized in terms of the history of evolution of the object base. For many practical applications like program synthesis plans, it is desirable that the system be capable of examining its state path and state space. Under the state space view, we want to be able to talk of particular state sequences, and sequences of operators that cause state changes. We call the former a *Transformation Sequence* which is a sequence of states of objects, and the latter *Rule Application Sequences*, which is the dual concept, i.e. sequences of rules that have been applied. We denote them respectively by ts and ras.

A whole gamut of other properties of interest of objects at least with respect to programming processes exist. These include data flow properties, which are essential for guiding program development and supporting it; object class descriptions, constraints, etc. In the case of rules one may be interested in things like information about objects or rules it can apply to; properties of objects it touches; effects such as what it can transform and into what can the transformation take the original node; its rule class, for example whether it is or not a metarule; whether it affects control in the system, etc.

In other parts of this section we have discussed means for denotation of property accesses. In the following paragraphs we introduce a notation for RASs and TS that will be used in other examples. TSs can be either *explicit*, when they are stored in the program description denoting the history of transformations of a node or when the sequence is explicitly given, and *implicit* when they have to be determined (like for example checking that applying certain rules would produce a TS of a certain kind). This corresponds to having a referent to a TS, and to computing the referent. To get the transformation sequence of a node, we access its transformation property via the structural property `TRANSFORM`. Thus, `*.transform` is a referent to the current node's TS. Testing TSs in general will be testing that they satisfy certain patterns. We include these patterns as schemas delimited by brokets.

```
< set .. list >
```

The elements of a TS can be object descriptions themselves. To test whether a certain node satisfies a certain transformation sequence we use the following notation:

```
''an object''.ts : < SET .. LIST >
```

Of course we can use this notation with node classes using the V mapping operators. Assume maps a predicate onto a set. Then we can have things like:

```
∧{N.ts:<set .. list> | N:list ∧ N.element:relation}
```

which will test whether all sets of relations have been transformed into lists of relations.

Testing properties of RASs is similar in nature to testing properties of TSs. The only difference currently is implementational in nature: we don't store explicitly the rule applications themselves but only their transformation sequences. As it is now, we are not storing the rule applications but the nodes as they are transformed by the rules. An application of TS and RAS in a PE is in planning and in developing search control heuristics. Their use can aid to plan state sequences that avoid particular paths, or operator sequences that have particular characteristics.

5.3.2 Operations on Process Objects

When talking about processes and in particular about the process of using CHI there are certain recurrent operations that need to be expressible. Among them we can see:

1. **Testing**: by which we check the validity of assertions about process and other objects

2. **Application**: by which certain functions are evaluated or through which rules and constraints are applied to objects.

3. **Constraining**: through which invariant relations in system organization are maintained.

4. **Modification**: through which objects are changed.

Any formalism for the expression of order of process must have facilities for expressing these concepts in the formalism. We will attempt to go step by step through these illustrating their use.

Testing

A basic operation is testing whether an object satisfies a certain predicate or not. It may be convenient to test for properties of process objects like for example testing that a certain node has had a transformation history that satisfies a certain pattern (i.e. that its TS satisfies the pattern), and similarly for RASs. The following are examples of how we might denote different kinds of tests: (we have already seen some examples of these in our examples on object attributes)

```
X:set                       X is a set
X ∈ {X:object | X:set}      X is a set
```

X's domain is a set of primitives

```
X.domain.element:primitive
```

S's size is less than L's range size

```
S.size < L.range.size
```

test node transformation sequences

X is a mapping transformed from a set

```
X:mapping ∧ ∃Y. X:transform=Y ∧ Y:set
X.ts : '< .. set .. mapping >'
```

test rule applications to a certain node

rule 31 applied first and rule 40 last

```
X.ras : 'RULE-31 .. RULE-40'
```

test properties of rules

```
R:metarule ∧ R.kind=agenda        R is an agenda metarule
```

R is a rule that refines into enumerable stored objects

```
refines-into(R) : 'enumerable ∧ stored'
```

R transforms objects into sets or lists
`transforms(R) : '{set list}'`
`transforms-into(R) : list` *same but only into lists*

Typical Tests in CHI

As we see above, the notation based on object, attributes and schemas is powerful enough to express constraints on processes given a suitable set of interpretable attributes in the domain of programming. In the above examples we see how the capability for referencing process-related objects introduces additional expressive power to the process control language.

The use of tests can be extended to the testing of data flow properties necessary for program development. For example, to remove all objects in a certain scope S which are not referenced, assuming that the system has a property called `referenced` which computes whether the variable in question is referenced or not, we can express it as

```
*:data ∧ ~referenced(*,S) => destroy(*)
```

Likewise to attempt parallel object representations of an object referenced in several places in the program description we may have a rule of the form

```
*.references.cardinality > 1 => apply AttemptParallelRep(*)
```

This same kind of scheme can be extended to other dataflow operations such as dead-variable elimination, in this case by checking where they are created and where they stop being referenced (i.e their last reference in the program description).

RAS patterns are denoted as schemas delimited by double brokets as follows

```
<< RULE.1 .. RULE.32 .. RULE.43 >>
```

where the rule names could have been replaced by any description of rule objects. It is possible to use the : operator to verify that a certain node satisfies a certain RAS. E.g.:

```
*: << RuleDescription 1 ... RuleDescription n >>
```

This statement would specify a test on the current node being operated on. RASs may be useful in knowledge acquisition by analogy: given a certain target piece of code and a RAS, we can test whether the same RAS is applicable to another region and apply it. Thus,

$$\begin{tikzcd}
N1 \arrow[r, leftrightarrow, "analogy"] \arrow[d, "RAS-1"] & N2 \arrow[d, "RAS-2 \text{ where } RAS-1=RAS-2"] \\
R1 & R2
\end{tikzcd}$$

Analogy and Use of RASs

Application

The next important operation is that of applying rules to system objects or to the program description. We denote it by the name `apply`. It takes a description of what to apply and where to apply it. The description may denote a set of rules or a single rule. The locus may be a region of the object base or a particular instance. Application allows us to force or preclude the application of certain rules, or of rule classes to a program description. In order to be able to refer to rules we need a way of expressing rule referents. There are three ways that can be directly used:

name: the rule is specified by a rule name, or the rule class is specified by a set of names. This approach is feasible for small rule bases, but is not useable in large knowledge base development efforts. It is simple and efficient, and the rule names may be mnemonic for their effect. Nevertheless, it is hard to express metalevel knowledge based just on this. Also, it introduces the problem of naming conflicts.

effect: at the refinement level (i.e. first order rules) rules can add properties to nodes or transform them. The effect of first order rules can then be denoted by either specifying a *set of constraints on properties* that a rule may add to a node, where the constraints can be arbitrary logical expressions, or by specifying a *set of constraints on the nodes* which the current node can be transformed into.

analogy: given a description of the effect of a rule, it specifies rules whose descriptions are analogous to it. This modality of application is beyond the scope of this section.

In order to achieve effective closure, we must be able to use the above three referent mechanisms for rules at any level. Thus we can effectively have rules that apply rules to rules. Some examples of the use of application follow

```
apply RULE-34                                  by name
apply {R:rule| transforms-into(R,atom]}        by effect
apply choose {x∈rules | accesses(*,size)}      rules that touch SIZE
apply {x:rule | refines(*,explicit)} to *
```

```
    apply choose {RULE3 RULE4} to X
```

The Application Operation

This operation can be used in rules to generate process constraining rules such as

```
        p(*) => apply SetRule.124
        q ∧ r => apply choose {x:rule|transforms-into(x,set)} to *
```

Some Metarules Using Application

Modification

Any object in the system (as a result of the constraint of closure) should be capable of modifying any other object, by either forcing predicates on the object to take certain logical values (for example when an object is constrained to be enumerable), or for attributes to take particular values (for example when we define the typical element of a set). In general, we may say that any object should be capable of modifying any attribute of any other object in the system. Notice that this means that rules could alter class descriptions. And that these changes could propagate to all instances extant at any moment in the system. Similarly, TSs can be modified although the modification of a node transformation sequence is an operation of unusual complexity. It means to alter the knowledge base in such a way that all possible transformation sequences that included the given TS will be changed to include the modified version. In turn, this requires modifying all rules where such a thing can happen, i.e. which would generate such TSs. Rules can also be modified by changing rule components, adding preconditions or actions, modifying inner rule applications, etc. Dataflow may be modified through a basically declarative program description modification. Some examples of modification are

modifying object properties
```
*.size ← 3
```

modifying object structure
```
X.body ← 'begin Y; Z end'
```

modifying rules, a possible edit command
```
R.transform-into ← 'set'
```
rule will transform nodes into sets

Modification of Objects

Constraints

The final operation in our group of control of order of process is *constraining*, an operation by which relations are forced to hold between system components. In a fully-closed system (i.e. a system where all system components are objects in the object base), a process is a navigation of a user or of the system itself among the space of objects. Such navigation is a sequence of operations on the object base. Since processes are sequences in time of operations on objects, then constraining or attaching deduction schemes to processes is equivalent to attaching them to operations and the objects they touch. We must be able to constrain any object in the system. Thus, nodes may be constrained to be refined in certain ways, or to have certain characteristics in their attributes. Attributes may be constrained to fall in certain ranges or to be applicable to restricted classes, TSs of objects can be constrained to fall into a certain pattern, RASs of objects may be constrained to disallow or enforce application of certain rules, rules may be constrained to satisfy particular properties which is very useful during acquisition. dataflow in procedural components can be made to have particular attributes, etc. In summary, we will allow constraining of the following kinds: 1) Constraining relations of an entity to other entities, and 2) Constraining structure and/or properties of an entity to satisfy certain criteria (this already constrains an entity to belong to a particular category). Here are a few examples of what we mean by constraining:

- If the current node is a set, transform it in such a way that it becomes a *bitmap*.
- Do not apply rule RANDOM.300 here.
- Apply either rule 344 or rule FOO.53 to this node.
- First apply some rule that produces a distributed object from this node, and sometime after apply any rule that will take it into some kind of LISP property list.
- If the current node is of such and such characteristics, then only apply rules that will take it into some enumerable lisp object.
- The following operation should be refined into a primitive operation without side-effects.
- The following function should be refined into a generator.
- Constrain two objects to have the same representation, such that whenever one of them is touched, the other is touched in the same manner.

Within our process as navigation metaphor, constraints limit the state space of the system by limiting the space of possibilities open at any one time for either user or system. For example, if we constrain a set to end up being transformed into a bitmap, any state not compatible with this (i.e. that would preclude transformation into bitmaps) must not be accessible. Notice that the system state is the set of values of attributes in the system since the system state is the state of the object base. Constraining the state space can then be either constraining the *values* of attributes or the *creation* and *deletion* of objects. Of course these two are somewhat related to each other since objects are values of attributes. In general constraints are expressed in the following way:

```
constrain <attribute> <relation> <description-for-value>
```

where usual relations are equality and set-theoretic relations, numeric relations, etc. The following are some examples of constraining

```
constrain *:transform.class ~= set                   never transform into set
constrain *.class = $b.prop.anotherprop.class        equality of classes
constrain *.ts = < .. set .. list >                  take through set to list
constrain *:transform.class ∈ {encapsulator data}    restrict transform

constraining implicit node properties
constrain X.ras = Y.ras
constrain X.element.class ∈ {set relation}

constraining the rule application process
constrain X.ras = << .. RULE-300 .. RULE-30 .. >>
```

Some Constraints in V

Constraints are usually implemented by demons placed on the objects and their properties using the deductive formalism. The question now is how to implement these in a simple efficient form. The system has as its task the detection of violated constraints and the prevention of these violations. A cumbersome approach is to have the system check after every single operation all constraints to verify that the operation has lead to a legal state. if not, then it takes corrective action. It would be better if the system were able to check some distinguished constraints and operations, only when necessary. It seems that it is very expensive to evaluate constraints every time. Also it is impossible for a system to detect that all possible breakdowns preserve constraints since the system cannot predict its environment or future interactions. There are always possible circumstances that can violate the software constraining mechanism, for example say hardware failures.

5.4 Programming Knowledge in CHI

One of the main components of CHI is the programming knowledge base. This base of rules and metarules generates most of the synthesis processes in the system by either providing at any particular state of program specification or transformation a set of possibilities of manipulation of system objects by the system itself, where system objects includes also the program specification, and by generating the possible sequences of exploration of possibilities during synthesis operation. Programming knowledge takes then the form of *declarative possibilities* for transformation and refinement, and *declarative constraints* for guiding the exploration of the synthesis space.

The programming knowledge base is used to codify efficiency, strategic and acquisition knowledge. It provides a vocabulary for accessing entities relevant to the synthesis task: program description nodes, program annotations, node transformation sequences, rule application sequences, rules and rule properties, data flow, tasks, agendas, and efficiency knowledge. In turn, CHI provides a class of metalevel operations that operate on these entities: transformation constraints, rule application, task generation, etc.

We have already seen examples of program construction rules. The: following examples illustrate some metarules in V. The show how particular kinds of process control rules can be expressed as part of CHI's programming knowledge base.

1. **Preference:** For example, consider the rule "When refining a block node in LISP prefer a refinement to progn". We can denote this rule in the following way:

   ```
       *:block ∧ language=LISP => constrain *.ts st *:< ..PROGN>
   ```

   An equivalent way of providing the constraint would be to constrain the *RAS* for *the node as follows*

   ```
   *:block ∧ language=LISP => constrain &=*.ras
                              st &:<< .. 'transforms-into(&, progn) >>
   ```

2. **Restriction on Usage:** where a particular object class is restricted not to be transformed into another object class. The rule is "If the only uses of a set are enumerations and element insertions, then do not refine it into a boolean mapping", *similar to rules from* (Kant, 1979).

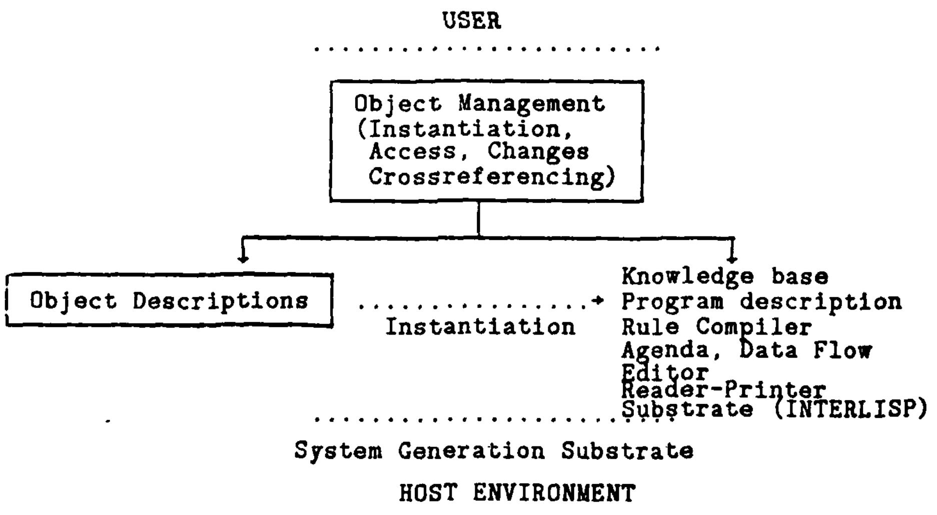
   ```
          *:set ∧ *.uses ⊂ {enumeration, insert-element}
            =>
          constrain &=*.ts st not &:< .. 'boolean mapping' .. >
   ```

3. **Constraints on Set Representations:** "If a set is used as the set argument of a membership test and if the only other uses of the set are enumerations and element *insertions, then refine the set to either* a *distributed mapping,* a *hash array or* a list".

   ```
          *:set ∧ *.set-of:∈
                ∧ *.uses ⊂ {member, enumeration, insert-element}
           =>
          constrain *.ts st & ∈ {< .. 'distributed mapping' ..>,
                                 < .. hasharray ..>,
                                 < .. list .. > }
   ```

§6 Implementation of CHI

There are some interesting theoretical and practical issues on the implementation of a system intended to be closed and self-described. It is difficult to produce a system that is completely closed with respect to its description in an initial implementation. CHI is no exception. The process we propose to introduce description closure is similar to that of bootstrapping a compiler. First, an initial implementation which provides basic closure machinery (like description and instance management, and self-reference) and descriptions for at least the programming parts of V is generated. This description can then be used to translate specifications of system components in V to the substrate (LISP) using a set of synthesis rules for the language. The next step is to write a rule-compiler (in the substrate language) to translating these rules into executable forms. These, plus a simple editor and reader/printer, constitute the first closure bootstrap step. We are currently engaged in developing the rule compiler and rule bases. The next step is to start defining segments of the rule compiler as rules and V procedures, and to provide rule-compilation object descriptions for agendas, demons, goals, etc. A similar process is done for the editor and other system tools. Eventually, all tools in the system as well as the objects they manipulate, are described in V and have executable counterparts in the substrate. At this point CHI has its own description expressed in the V language. Once this is done, all the machinery that has been developed can be used for maintaining, modifying and extending the system itself. Moreover, the system's own knowledge about programming can be used for support during loss of transparency in system-user interaction. In its end state, the system consists solely of instances of descriptions that have been translated into the generation substrate. The structure of a closed CHI system would be as follows:



CHI's Description Closure

In summary, after complete closure, the system is self-referential: all objects in the system are accessible from any system component on a uniform basis, and any object in the system is an instance of some other object which describes its structure and organization. We have implemented a pilot CHI system in this manner, which is currently being used to test our ideas. This initial version provides a closed object management base upon which we have built a reader/printer/editor subsystem which performs the conversions between internal and external form, and a rule compiler which translates synthesis rules into the language substrate. With these CHI currently provides capabilities for the management and access to object descriptions and instances, instantiation and specialization support, a user interface that includes facilities for input and output of system objects in external (*pretty-printed*) form, an extensive support facilities in the form of system-wide object and instance cross-reference, and a context mechanism for search control. The user printout interface characteristics are user-specifiable. There are currently different modes for translating internal to external structure, dependent on the view the user wants to ascribe to the objects being printed. This allows the user to manipulate system objects in the user's cognitive space.

§7 Self-Description and Self-Modification

Programming rules play an important role during the generation or compilation of CHI itself. The user of the system when specifying the rule space, specifies the possible set of implementations (i.e. structures) for program descriptions and thus for CHI itself. The rules are expressed in V, and the output produced by the system's rule compiler is efficient LISP code or code in the substrate language. As a postlude to a discussion on CHI, we present in this section an exploration of a possible compilation scenario in which the rule compiler compiles itself. Lets first look at the duality between rules and procedures in CHI (in a sense we want to motivate by doing this the idea that it is possible to write a rule compiler as rules).

Consider the rule

> "All rules that transform loops must save the transformed nodes in a set associated with the rule, via mapping G"

We may express this rule as program code in V, as follows

```
forall R suchthat
        R.class=rule ∧ R.kind=transform
                     ∧ R.referent.class=loop
  do add-part(R.consequent, add-part(G(R),R.referent))
```

As a V rule, it would be expressed as

```
*:'transform rule' ∧ *.referent.type=loop
  =>
add-part(*.consequent, add-part(G(*),*.referent))
```

Another example has to do with forcing certain program construction paths (property attachment or transformation) during system use. Consider the rule,

> "All sets in the program description should be transformed into boolean mappings"

Again, as V program code the rule would look

```
        forall S suchthat S.class=set ∧ S.use=program
           do block
                OldS ← S;
                transform(S, mapping);
                S.domain ← OldS;
                S.range ←  { True, False }
              end;
```

and as a rule,

```
                transform *:set ∧ *.use=program
                  =>
                mapping from * into  { True, False }
```

We will now proceed to show an example in CHI of the use of closure and self-description to change a system component.

7.1 Closure in the Rule Compiler

The rule compiler in CHI takes rules in V and transforms them into executable forms in LISP. We will attempt to give to the reader an idea of the process by which closure may be achieved in the rule compiler, by expressing the compiler itself as rules. For the purposes of this example, assume rules take the form

```
a1 ∧ a2 ∧ a3 ... ∧ an => g
```

i.e. an antecedent formed by a conjunction of predicates and a consequent which is some actions to be performed. We will assume that available to any rule at the moment of its application is the object it is being applied to, and that that object may be referenced as * in the body of the rule. Assume the description for a rule is schematically

```
rule
  antecedent : set of conjunct
  consequent : V code
```

i.e. it consists of an antecedent which is a set of conjuncts (which in turn are predicates) and a consequent which is a piece of V code. Also assume that there is a V control object called a `block` (identical to its analogue concept in ALGOL-like languages), and another control object called `vblock` which is a value-returning block (similar to `progn` in LISP, or the block construct in ALGOL-W). We will assume that a block has the description

```
block
  steps : sequence of V code
```

Rules in this example are transformation rules, that is, rules that check conditions on some object and replace it by another which is produced by the consequent of the rule. A possible rule compilation rule that compiles rules into V (which can then be transformed into LISP using CHI) is shown in its external form (printing form) below:

```
transform 'transform a1 ∧ a2 ∧ ... an => g'
 =>
'block
  achieve(a1);...; achieve(an);
  transform(*,g);
end;'
```

where we have assumed the existence of two V procedures called **achieve** which attempts to verify whether an assertion holds (perhaps even by suspending a demon waiting for more information to come), and **transform** which takes two argument objects and transforms (i.e. physically replaces) one of them by the other in the object base. The internal representation for that particular rule is:

```
transform *:rule ∧ *.kind=transform ∧ *.antecedent
                         ∧ *.consequent
   =>
vblock
  @ ← make-object(block)
  @.steps ← {'achieve(m)' | m∈*.antecedent}
  @.steps ← Union(@.steps; 'transform[*,*.consequent]');
  @;
end;
```

Basically what this rule says is that to translate a transformation rule then you must produce an object which is a block, which achieves each conjunct in the rule in sequence, and which transforms the focus object of the rule (i.e. *) into an object constructed by the consequent of the rule. Note that the rule compiler rule shown above is an instance of one such transformation rule. If we have bootstrapped the system so as to have a first running version of the compiler, it is possible to have this rule compile itself. For assume we apply it to itself. Then it generates a block, whose steps are to achieve the test that the current focus is a transformation rule, and a transform operation that creates the rule translation. Thus, the rule compiler transformation compiling rule will transform itself into

```
block
  "hereby check rule is transform rule"
  achieve[*:rule]; achieve[*.kind=transform];
  achieve[*.antecedent↓null]; achieve[*.consequent↓null];
  "if we succeed then we transform the rule into a block"
  transform[*, vblock
                 @ ← make-object(block)
                 @.steps ← {'achieve(m)' | m∈*.antecedent}
                 @.steps ← Union(@.steps; 'transform[*,*.consequent]');
                 @;
               end;];
end;
```

Notice that now the rule itself has disappeared as rule, and is now normal V code. This code is analyzable by CHI, may be manipulated as an object, and can be translated into the host environment in an efficient manner using CHI. Thus, for example, if we add a rule that provides a more efficient representation for sets under certain circumstances, we may use CHI to recompile all rules in the system so that they run more efficiently, and CHI itself will thus run faster!

§8 Related Work

The search for adequate programming tools is as old as computers themselves. The early 60's mainstream research concentrated on the development of programming languages and compilers. In 1966 Warren Teitelman published a thesis on a system called PILOT which could couple easily to its users (Teitelman, 1966). Almost all of the work described in his thesis carried over to the INTERLISP programming environment. Later efforts like the Reactive Engine (Kay, 1970) and Copilot (Swinehart, 1974) attempted to solve problems of user-computer coupling in other contexts. In the 70's a major shift in emphasis towards disciplines for systematic programming occured. These disciplines translated into a quest of tools for logical and precise program development (e.g. the PASCAL effort). In parallel, work on program synthesis tools started thereby merging AI research on programming with other work. Three major system implementation efforts were undertaken: PSI (Green et al, 1979), SAFE (Balzer et al, 1976) and the Programmer's Apprentice (Rich et al, 1978). In the last few years new research directions have started on the application of techniques used in LISP programming environments to more conventional programming languages like PASCAL or ADA.

Other related work we might mention in the context of CHI includes the work by Davis on metalevel knowledge (Davis, 1976) which explores in depth the use of rules in the guidance of processes, the work by Schwartz and his group on set-theoretic programming languages (Kennedy and Schwartz, 1975) as a device for concise very high level specification of programs, and the work of Barstow on the use of rules for codification of programming knowledge (Barstow, 1979). For a survey of artificial intelligence research on automated program construction the reader is referred to (Elschlager and Phillips, 1979).

§9 Conclusions and Summary

In this section we have explored the notion of a knowledge-based programming environment. Recent knowledge-based automatic programming research has focused primarily on the task of program synthesis. Here we have extended the horizon of this research to include a broader part of the programming process, i.e., the programming environment as a candidate for artificial intelligence application.

We have argued that the utility of a smart programming environment is enhanced by a self-descriptive capability. One assumption is that the aspect of programming that is of interest is programs undergoing change, either in the form of creation, modification or maintenance. A second assumption is that the making of changes in target programs is often facilitated by making appropriate changes in the programming environment. An example is where a text editor (part of the environment) must be extended to edit a new data structure (part of the target program). The tools provided by the programming environment can more easily assist in this self modification process if the environment is itself described in terms that the tools for modification can deal with.

We have suggested the utility of such a knowledge-based, self-described programming environment through the presentation of a series of scenarios. Some of the scenarios are within the scope of current technology while others suggest future research directions. A more detailed investigation of both the feasibility and utility of such environments seems well-warranted by the glimpses given herein.

Next, we have explored the ideas of self-description and knowledge based environments in a more concrete way. We describe a particular system, CHI, that embodies some of the key features that are within reach of current technology. CHI is described by a single language, V, for specifying not only programs but programming knowledge (usually in the form of rules) and also itself. Portions of the V language are illustrated by examples showing how in V we can represent (1) high level program specifications (2) program synthesis rules or facts (3) higher-level program synthesis meta-rules or plans (4) constraints on program consistency and (5) CHI's own agenda mechanism. The expressive power of the language has been satisfactory, and quite an improvement over earlier languages for synthesis rules. We conclude that a practical language for such purposes is feasible, with the V language showing one possible approach.

Finally, we have discussed how the V language may be built on an object-oriented data management (or knowledge management) system that allows uniform access to the information needed by the programming environment.

A version of the CHI system and the V language has been completed and further extensions to the implementation are in progress. Versions of the knowledge base manager, the rule compiler, the editor, as well as many synthesis rules now work, and have been tested on simple program synthesis tasks. Although many significant research problems remain, it appears that the path proposed may well lead to more sophisticated programming environments.

This section has benefited from the interaction and help of Steve Westfold, Steve Tappel, and Tom Pressburger in the development and implementation of our ideas. The authors would like to thank them, as well as Beverly Kedzierski, Jerry Feldman and Sue Angebranndt for very helpful comments on content and presentation. Jorge Phillips would like to thank Fernando Flores for innumerable conversations and insights on cognition and the nature of embedded computer systems, and for having introduced him to the works of Martin Heidegger and of Humberto Maturana. His ideas and those of the latter authors have had a strong influence in one way or another on the final shape of this work.

§10 References

Barstow, D. **Knowledge-Based Program Construction.** The Computer Science Library, Programming Language Series. Elsevier-North Holland Inc. New York, 1979.

Balzer, B, Goldman, G., and Wile, D. *On the Transformational Implementation Approach to Programming,* **Proceedings, Second International Conference on Software Engineering,** Computer Society, Institute of Electrical and Electronics Engineers, Inc., Long Beach, California, October 1976, pages 337-344.

Bobrow, D. and Winograd, T. *An Overview of KRL, a Knowledge Representation Language,* **Cognitive Science,** 1977, I, pp 3-46.

Elschlager, R. and Phillips, J. *Automatic Programming,* Memo HPP-79-24, Report STAN-CS-79-758, Heuristic Programming Project, Computer Science Department, Stanford University, Stanford, California, November 1979.

Green, C. *The Design of the PSI Program Synthesis System.* **Proceedings, Second International Conference on Software Engineering,** Computer Society, Institute of Electrical and Electronics Engineers, Inc., Long Beach, California, October 1976.

Green, C., Gabriel, R.P., Kant, E., Kedzierski, B., McCune, B., Phillips, J., Tappel, S., and Westfold, S. *Results in Knowledge Based Program Synthesis,* **IJCAI-79: Proceedings of the Sixth International Joint Conference on Artificial Intelligence,** Volume 1, Computer Science Department, Stanford University, Stanford, California, August 1979, pages 342-344. included as section 10.

Kant, E. *Efficiency Considerations in Program Synthesis: A Knowledge Based Approach,* Ph.D. thesis, Memo AIM-331, Report STAN-CS-79-755, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, Technical Report SCI.ICS.U.79.1, Computer Science Department, Systems Control, Inc., Palo Alto, California, September 1979.

Kay, A. *The Reactive Engine*. Ph.D. Thesis. Computer Science Department. University of Utah, Salt Lake City, Utah. 1970.

Kennedy, K. and Schwartz, J., *An Introduction to the Set Theoretical Language SETL*, in **Computers and Mathematics, with Applications**, Volume 1, Number 1, 1975, pages 97-119.

Masinter, L. *Global Program Analysis in an Interactive Environment*, Ph.D. Thesis. Stanford University, Stanford, California, 1980

Phillips, J. *Self-Described Programming Environments: An Application of a Theory of Design to Programming Systems* Ph.D Thesis, Stanford University, Stanford, California.

Rich, C., Shrobe, H., Waters, R., Sussman, G., and Hewitt, C. *Programming Viewed As an Engineering Activity*, AI Memo 459, Artificial Intelligence Laboratory, MIT, Cambridge, Mass. January 1978.

Stallman, R. *EMACS, the extensible, customizable, self-documenting display editor.* AI Memo 519, Artificial Intelligence Laboratory, MIT, Cambridge, Mass. June 1979.

Stefik, M. *An Examination of a Frame-Structured Representation System.* **IJCAI-79: Proceedings of the Sixth International Joint Conference on Artificial Intelligence,** Volume 1, Computer Science Department, Stanford University, Stanford, California, August 1979.

Swinehart, D. *COPILOT: A Multiple Process Approach to Interactive Programming Systems.* Ph.D. Thesis. Computer Science Department. Stanford University, Stanford, Ca. August 1974.

Teitelman, W. **INTERLISP Reference Manual.** XEROX Palo Alto Research Center, Palo Alto, Ca. December 1978.

Winograd, T. *Beyond Programming Languages.* **Communications of the ACM,** p361-ff. July 1979.

Section V

On the Use of Self-Description for Knowledge Acquisition

Jorge Phillips
Cordell Green
Steve Westfold

Abstract: Knowledge acquisition, or the addition of new capabilities to large, complex (artificial intelligence) systems is a critical problem. This section describes a possible solution to this problem. A knowledge-based programming system may be described in its own terms so that the problem-solving tools it provides may be used to support its own augmentation and reprogramming. Knowledge acquisition in such systems may be considered to be the same as system extension or system building. Some of the ingredients discussed for supportive knowledge acquisition are a system-accessible system description, a language capable of useful self-description, an appropriate knowledge base, and a set of tools built out of these components and adequate for operating on them. These ideas are examined in the context of the CHI knowledge-based programming environment but may be applicable to other AI systems.

§1 Introduction

A critical problem in artificial intelligence is knowledge acquisition. Knowledge acquisition is one view of the process of extending the capabilities of intelligent systems. System extension can take many forms, from automatically learning new rules, to being told new rules, to just "ordinary" reprogramming. One can also view all of the processes by which system capabilities are extended as some form of reprogramming, be it in a rule language or whatever.

We will explore the idea of viewing the many processes of system building as programming activities. Programming is more than just writing programs; it includes the debugging and development of the system. Take as an example, a rule-based system. The development of a rule-base entails editing, debugging, and cross-referencing the rules; compiling the rules into an efficient form; storing and retrieving (managing) rules in rule bases; instrumenting the rules to measure performance; printing the rules in new formats; testing the rules for consistency, managing interactions between the rules, etc. Each of these operations has its analogue in any programming system. More generally, one can have high-level languages not only for rules, but also for the rest of the programming environment for a rule-building system. The same argument may be applied not only to systems with knowledge expressed in rules but also to systems with knowledge expressed in logic, frames or procedures.

Given that much of system building and knowledge acquisition can be viewed as various programming activities, can we employ the methods of knowledge-based programming to assist in these activities? We believe the answer is yes; an AI system can profit from being built on top of a knowledge-based programming environment which includes a description of the AI system itself. Our belief is based on our experience with the development of the CHI knowledge-based programming system (Phillips) (Phillips and Green) which we will discuss as an embodiment of our ideas on system building. The results presented on self-described programming environments are primarily a summary of Jorge Phillips' Ph.D dissertation research. A subset of CHI has been implemented and is currently in the process of being extended by the authors and other members of the CHI project. In the rest of this section we discuss CHI as envisioned. The current state of implementation is described at the end of section 3.

§2 Supportive Knowledge Acquisition

In this section we show what we mean by support during knowledge acquisition by presenting four typical situations in which a knowledge-based (KB) system can be supportive during system development. We mention briefly here how a self-description enables a system to deal with these situations, and we compare the situations to analagous ones encountered during programming. Later on we discuss the architecture of CHI, a self-described KB programming environment.

2.1 Adding Rules to a KB System

Consider the addition of a new piece of knowledge such as a rule. The system should have descriptions of what are valid system states in the form of constraints. The new rule can be examined to see whether its application could lead to violation of these constraints due to undesired interactions with certain other rules. If so, the system could assist the user in making the necessary modifications to ensure the constraints will not be broken, e.g. by restricting rule applicability or adding checks in the rule body. The problems in adding knowledge to a system are comparable to those often encountered in programming when modifying a procedure, e.g. changing a procedure to do something desirable in one context may cause it to behave incorrectly in some other contexts.

2.2 Internal Errors in the System

Now consider what happens when a KB system has an error while performing some task. The system uses the description of itself to find which aspects of its state are inconsistent (violated constraints, missing rules, etc.) and reports these to the user. It then guides the user through the inconsistencies which the user fixes, and it verifies that consistency is restored. The user may notice that similar errors are likely to occur again and that similar fixes would be appropriate, so the user guides the system in generalizing the fix and providing a suitable applicability test. Additionally the user may ask the system to present those components of the system which may have created the error, and thus find the initiator of the inconsistent state. In normal programming, this kind of activity would be very smart debugging.

2.3 Retrieval of Relevant System Objects

For the third example, imagine the user is focussed on a particular subgoal and wants to know what the system can do to solve it. The system can describe the objects (rules, methods or plans) it has for achieving the subgoal and it may present the changes that each would make in achieving the subgoal. The user can then select the appropriate method. In programming this could be conceived of as a smart development assistant.

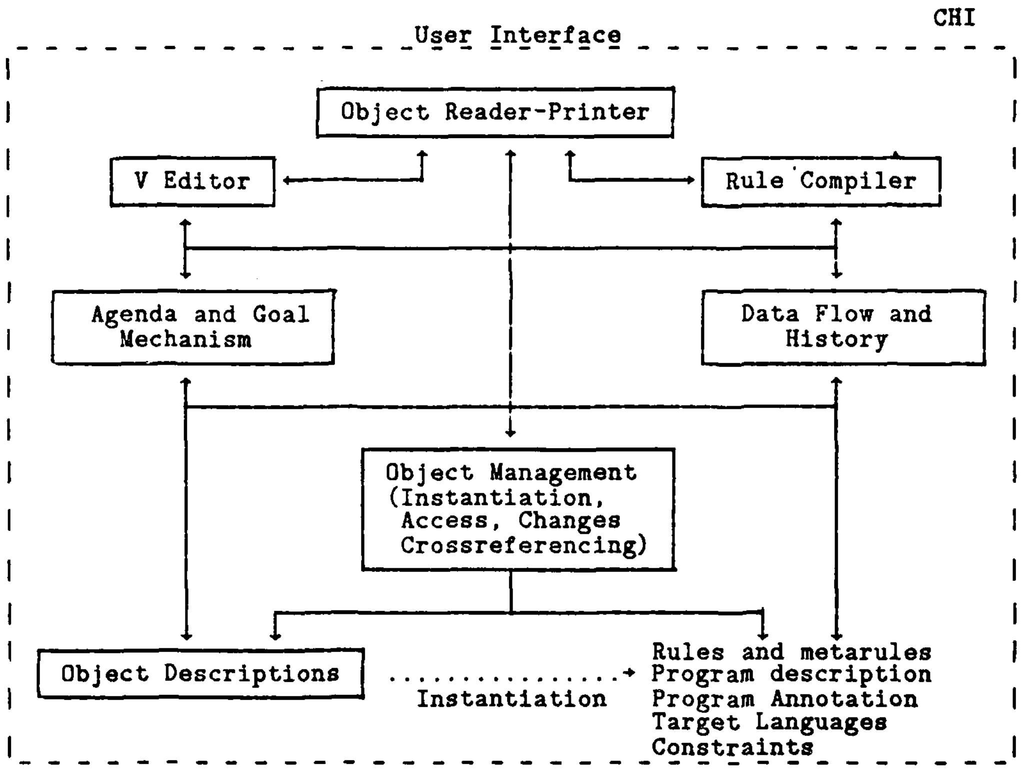
2.4 Modifying the System

Suppose the user wants the system to use less space. The system could examine its state to find where it uses a lot of space. Suppose that much space is used for storing descriptions. The system could determine a more space-efficient data structure, possibly with the help of new data structure rules provided by the user, and recompile all its description access functions to take advantage of this data structure. The system would also convert its description data base to the new format.

The preceding examples have something in common: they point to situations where the system is not performing as desired or intended, and where the system provides support for repair by using its own description. We view knowledge acquisition then as coupling of the system to its environment, via system-mediated expertise transfer from users. All these instances of support can be viewed as instances of supportive knowledge-based programming.

§3 CHI: An Experimental Knowledge-Based Programming Environment

CHI is a self-described programming environment, and may be viewed as a collection of KB tools operating on KB program descriptions. These tools are built on top of a uniform interface which provides homogeneous access to a knowledge base of UNIT-like objects (Stefik, 1979). These objects are grouped into specialization, inheritance, and instance hierarchies which reflect the structure of the system's application domain. The figure depicts the general organization of the system.



The CHI System: Initial Design

CHI provides the user with capabilities for the management of system objects. These include creation, deletion, instantiation, and specialization of objects such as descriptions, system properties, rules, etc. The user interface provides facilities for working with the knowledge base and system objects in a user-oriented form. The system also provides extensive support facilities in the form of system-wide cross-reference, a context mechanism for hypothesis testing, and a simple rule compiler which translates rule descriptions to the host language which in this case is INTERLISP. CHI's knowledge base has general information about programming: refinement, efficiency analysis, printing internal structures in readable form, editing, well-formedness of programs, semantic consistency of programs, cross-referencing, data flow analysis, etc.

Knowledge acquisition in such a system ranges from the development of rule bases and descriptions for concepts the system manipulates, to the development of tools that operate on these. The following are typical instances of the use of CHI to develop its own knowledge base. They provide concrete examples of the capabilities we are envisioning that self-description may give to KB systems.

- Synthesis of efficient program transformation programs from concise high-level rules (Rule Compiler).
- Synthesis of efficent program consistency tests from high-level rules.
- Support during refinement. For example, user may ask for all rules that refine sets into particular data structures.
- Self-improvement. For example, rules for a new data structure are introduced, and CHI then recompiles itself using the new data structure where appropriate.

The current status of the implementation of CHI is that all the components shown in Fig. 3.1 are present, at least in preliminary form. CHI has successfully been used to synthesize several test programs.

CHI is our starting point for extending or rebuilding new systems. We shall speak of two aspects of such a system: the language we use to describe and interact with the system, and the set of capabilities and tools available.

§4 The language

The idea of self-described KB systems is centered around a uniform language in which the user describes system components and the processes that operate on them. This language is used to describe the system itself in a manner which is accessible to the system. CHI's description language is called "V". We will summarize the main ideas in the language. The interested reader is referred to (Phillips) for a more detailed exposition. It is convenient to program in terms of operations on set theoretic structures (or suitable views of them) like sets, mappings, relations, records and their compositions on primitive and structured objects, via manipulations that create, add or delete components from structures. The concepts of V range from low level concepts like lists to process related concepts like transformation histories and rule application sequences, needed for dealing with the evolution of the system along time. A system-accessible description of the language semantics and the organization of the system is useful as a tool for management of complexity, by providing both consistency constraints and what could loosely be termed *domain support.* V is extensible, i.e. it allows for continuous change and development of the objects used in program synthesis, and of the programming concepts it employs. The most important aspect is that the language constructs capture the most fundamental concepts the user wants to deal with.

The following are simple examples of V:

A. Objects are described in terms of selectors and predicates

`F.domain.element:set`	*F is a map with domain a set of sets*
`A:'partially ordered agenda'`	*A is an agenda implemented on a poset*
`A:(*.ordered ∧ *.size=3)`	*A is ordered and of size 3*

KB assertions and queries in V

B. A Simple Agenda in V

The following code shows how a simple agenda is expressed as a structural computation on high level objects, in this case a set of tasks.

```
module AGENDA
  entries DoTask, CreateTask;
  type task = plex
                node : object;
                attribute : name
              end;
  var Q : set of task;

procedure CreateTask (O :object; N : name);
Q ← Q ∪ {create(task, node=O, attribute=N)};

procedure DoTask;
select X∈Q, R:rule st applicable(R,X!node)
   do apply(R,X!node);

end AGENDA;
```

A Hypothetical Agenda Described in V

C. Refinement Rules, Metarules and Constraints

Some examples of typical rules are:

1. **A representation of a set:** "A set may be represented by its characteristic function, as a mapping into boolean". This rule is written in V as

   ```
   transform A:'set of X' → A:'mapping from X to boolean'
   ```

2. **A transformation of a membership test into an existence test:** "A membership test on a stored collection may be refined into a test on whether any item in the collection is equal to the item being tested". This can be denoted by

```
    transform A:'x∈S' ∧ S:'stored collection' → A:'test ∃y∈S st y=x'
```

Here are some simple constraints that restrict the class of the transformation of some object and others that restrict the sequences of rules that can be applied (ras is rule application sequence):

```
constrain *.transform.class ~= set        never transform this node into a set
constrain X.ras = Y.ras                   same refinements on X and Y
constrain X.ras ↓ << .. RULE-300 .. RULE-30 .. >> avoid some path
```

Some Constraints in V

Here is a simple metarule that activates constraints.

"If the only uses of a set are enumerations and element insertions, then do not refine it into a boolean mapping". We can denote this rule in the following way (ts is *transformation sequence*):

```
*:set ∧ *.uses ⊂ enumeration, insert-element
  =>
constrain &=*.ts st not &:< .. 'boolean mapping' .. >
```

§5 Tools and Capabilities

The programming tools in CHI profit from the system's self-description, uniform access to objects, and integration with the other tools in the system. Such tools include a reader, printer, editor, debugger, program synthesizer, function libraries and a consistency maintainer. Because the tools are driven by descriptions, they are applicable to any part of the system for which there is a description. This helps to avoid a proliferation of tools, each with a slightly different user interface. For example, the system-wide edit focus changing and printing commands can be used for such things as exploring the program-transformation history. Here we give some examples of problems that arise and their interaction with the system structure.

What are the consequences of editing in a system that tries to maintain consistency? Typically editors provide a basic set of simple commands which can be composed to produce any desired change. Most of these commands do not preserve consistency. The system can note the inconsistencies as they occur but should delay complaining about them because in most cases the immediately following edit commands will restore consistency. Restoration of consistency can be detected by setting up demons or reanalyzing the altered objects. The system must face the problem of how long it should wait before complaining about an inconsistency. One promising technique for minimizing this problem is to group the simple alteration operations into macro operations that as a whole maintain consistency. Such edit macros share much in common with CHI transformation rules. In particular, much of the system machinery for dealing with rules can be applied to these "edit rules", including their acquisition, compilation and indexing. The obvious problem with this approach, though, is that with the large number of edit rules it will be difficult to remember the name of each one or even whether the needed one exists. What is needed is a sophisticated rule-reference language.

An interesting consequence of having an editor which can edit any system object is that it can be applied to parts of the system not normally considered alterable, provided there is adequate description of the objects altered, in particular, what constraints they are involved in, and what actions are necessary to maintain these constraints. It is conceivable, for instance, that the user could edit the rule transformation sequence, replacing one rule applied by another. It should not be difficult for the system to determine what has changed at the point immediately after the rule application, but then it needs to trace the dependencies of these changes through later rule applications, making changes as necessary. It is quite possible that the system will find that some later rule is no longer valid, but it may still be able to salvage things, probably under user control with system guidance.

§6 Conclusions

This section has shown a possible approach to supportive knowledge acquisition based on the use of self-description. It discusses how self-description is closely tied to the use of a uniform language for interacting with and describing the tools provided by the system. In this manner a self-described KB programming environment may advantageously use its own problem-solving tools to support its own augmentation. This approach seems promising as a new method for supportive engineering of large AI systems, where the AI system is implemented on top of a self-described KB programming environment. One such environment, CHI, is studied in this context. A pilot version of the system has been implemented and is currently being used to test the ideas presented in this section.

§7 Acknowledgements

The authors would like to thank other members of the CHI group, in particular Sue Angebranndt, Beverly Kedzierski, Tom Pressburger and Steve Tappel, for discussions leading to these results and for providing a conducive environment for this research.

§8 References

Barstow, D. **Knowledge-Based Program Construction**. The Computer Science Library, Programming Language Series. Elsevier-North Holland Inc. New York, NY 10017. 1979.

Green, C. *The Design of the PSI Program Synthesis System.* Proceedings 2nd International Conference on Software Engineering, October 1976, pp 4-18.

Green, Cordell and Westfold, Stephen *Knowledge-Based Programming Self-Applied, Machine Intelligence 10,* Ellis Forward and Halsted Press (Wiley), 1981 (forthcoming).

Phillips, Jorge *Self-Described Programming Environments: An Application of a Theory of Design to Programming Systems.* Ph.D Thesis, Electrical Engineering and Computer Science Departments, Stanford University (forthcoming).

Phillips, Jorge and Green, Cordell *Towards Self-Described Programming Environments,* this volume, section 5.

Stefik, M. *An Examination of a Frame-Structured Representation System.* Proceedings Sixth International Joint Conference on AI, Tokyo Japan, pp. 845-852. 1979.

Teitelman, W. **INTERLISP Reference Manual.** XEROX Palo Alto Research Center, Palo Alto, Ca. December 1978.

Section VI

Summary of Algorithm Design

This section presents a summary of progress in Algorithm Design.

We use the term *algorithm design* to refer to the more complex and creative aspect of the programming process in which new algorithms are designed, modified and debugged. This part of programming may be contrasted to the more straightforward aspects such as data structure selection (though of course a well-defined boundary is difficult to draw). In our other ARPA project, "Knowledge-Based Programming", we have established a knowledge-based approach for the development of a computer program, called CHI, to assist in these straightforward parts of programming. In the algorithm design project, we are extending the knowledge-based approach into more difficult areas in order to develop an intelligent set of tools for algorithm design. Thus this research may be viewed as an extension of research on the CHI system.

The key to success in this research is finding a set of principles that underly the algorithm design process. A priori there is little evidence to indicate that algorithm design is an orderly process. Perhaps the most positive evidence is the belief that computer scientists can be taught some very general principles of design. Accordingly, modern textbooks include discussion of a few general principles. But there has existed no formal computer-amenable theory in this area.

Our first year of effort to formalize and codify algorithm design principles has yielded results that are rather surprising, in that algorithms that appeared very dissimilar on the surface were found to be derivable by using the same principles. We began our study with a set of different combinatorial algorithms of reasonable difficulty, including several versions of prime-finding algorithms ranging from the simplest to recently discovered linear-time versions (plus one new version discovered in our studies) as well as several shortest path algorithms including dynamic programming versions. The primes algorithms seemed quite different from the shortest path algorithms. The results we obtained showed that there are derivations of every one of these algorithms based upon the same set of methods, the chief one of which we call the operator incorporation method. In this method the algorithm is specified very simply, usually as a generate and test construct. Then efficient versions of the algorithm appear as successive refinements of the specification where the tests are incorporated into the generator.

As a simple illustrative example, consider an "even squares" algorithm that finds the even perfect squares less than some bound. The program may be specified quite naturally as the set of natural numbers that are even, square and less than the bound. This high-level program/specification can be thought of as a generator of natural numbers combined with a test that only passes the even squares. Then, various versions of the algorithm are derived by incorporating the even test and the square test into the generator so that candidates are never generated, rather than being eliminated after they are created. For example, rather than generating each number and testing to see if it has an integer square root, the square test is incorporated into the generator so that just the set of squares are generated by multiplying each number by itself. Thus the program is reduced from a linear-time version requiring an expensive operation at each step, to a square-root time version requiring only an addition and multiplication at each step.

Operator incorporation is not the only method, as it must be combined with simplifications and other transformations, but it has been the key technique. It has turned out that the principles we actually developed during this first contract period are rather different than those proposed at the start of the period. The new principles have less overlap amongst themselves since the information is better factored. Three principles became different instances of the operator incorporation method. In addition, we developed a process graph formalism to represent the principles and a non-deterministic stream semantics that allows a precise definition for the formalism. A short paper that uses the process graph formalism to illustrate the design of a shortest path algorithm is included in section 7.

Using the operator-incorporation methods, we have also shown how a prime-finding algorithm can be derived that runs in linear time, compared to polynomial time to directly interpret its high-level specification, and how shortest path can be cut from exponential to quadratic time. To further establish the generality of this methodology, we then looked at a wider variety of algorithms to see if they could also be derived by the same principles. The results, were that derivations for the majority of those surveyed fell within the scope of the operator incorporation method. The next most commonly used method was the divide-and-conquer principle. We have included as section 9 a draft of a description of the divide-and-conquer method and a detailed derivation of a binary search algorithm using this method.

At first, the algorithms we have studied might seem too theoretical to be of practical consequence. But in fact they are closely related to real-world algorithms with high payoff. For example, during the process of refining our algorithm design principles we discovered a new concurrent shortest-path algorithm. For most cases of interest it runs in order $n \log n$ time versus n^2 for the sequential version, which is quite significant for large problems. In addition the algorithm lends itself to a VLSI implementaion on a tree machine. An interesting fact is that the problem it solves is also the cruise missile path optimization problem. By using massive parallelism our algorithm runs much faster than the one now being implemented. In future cruise missile developments our algorithm would be one of the candidates to be evaluated for VLSI implementation in an on-board controller. Whether or not this particular algorithm has the right properties for that problem is not known, but it is clear that the tools we are developing will be of more than theoretical interest.

In addition to the results discussed above, we have taken another major step toward our long-term goal of building practical tools for algorithm design. The results mentioned so far have indicated the feasibility of the long-term goals, but at the pencil-and-paper study level. The real proof is in building a complete system and testing it on real world problems.

In the first contract period we took the first step and implemented a test algorithm design system that demonstrated the soundness of the approach. Two interesting results emerged. First the principles had indeed been defined well enough to be implementable, though not without further refinement. Second, we were able to extend the CHI knowledge-based programming sytem to be the vehicle for the implementation of the algorithm design system. This was our hope and it shows the extensibility of our program refinement framework. The result is a more effective tool that encompasses both the more difficult algorithm creation aspect of programming and the more straightforward parts such as data structure selection. Of course, both ends of the spectrum are necessary parts of the general programming process and integrated design tools will allow the maximum payoff.

Our implementation has been tested on the design of a first algorithm, namely the even squares example described above. This example problem has been an excellent vehicle for testing and refining both the implementation and the necessary algorithm design knowledge. The even squares problem is such that by varying the order of constraint incorporation several versions of the algorithm are achievable. Section 8 of this proposal is a detailed discussion of the derivation of an even squares algorithm, showing the exact form of both the program specification and the transformation rules.

Section VII

Some Algorithm Design Methods

Steve Tappel
Systems Control, Inc.
and
Computer Science Department, Stanford University

Algorithm design may be defined as the task of finding an efficient data and control structure that implements a given input-output specification. This paper describes a methodology for control structure design, applicable to combinatorial algorithms involving search or minimization. The methodology includes an abstract process representation based on generators, constraints, mappings and orderings, and a set of plans and transformations by which to obtain an efficient algorithm. As an example, the derivation of a shortest-path algorithm is shown. The methods have been developed with automatic programming systems in mind, but should also be useful to human programmers.

Many helpful ideas and criticisms were provided by Cordell Green, Elaine Kant, Jorge Phillips, Bernard Mont-Reynaud, Steve Westfold, Tom Pressburger and Sue Angebranndt. Thanks also to Bob Floyd for sharing his insights on algorithm design.

§1 Introduction

The general goal of automatic programming research is to find methods for constructing efficient implementations of high-level program specifications. (Conventional compilers embody such methods to a very limited extent.) This paper describes some methods for the design of efficient control structures, within a stepwise refinement paradigm. In stepwise refinement (see for instance [1,2]), we view the program specification itself as an algorithm, albeit a very inefficient one. Through a chain of transformation steps, we seek to obtain an efficient algorithm.

$$\text{Specification} \mapsto \text{Alg} \mapsto \ldots \mapsto \text{Algorithm}$$

Each transformation step preserves input-output equivalence, so the final algorithm requires no additional verification.

This paper is substantially the same as one that appeared in the Proceedings AAAI Conference, Stanford, 1980.

Algorithm design is a difficult artificial intelligence task involving representation and planning issues. First, in reasoning about a complicated object like an algorithm it is essential to divide it into parts that interact in relatively simple ways. We have chosen asynchronous processes, communicating via data channels, as an appropriate representation for algorithms. Details are in Section 2. Second, to avoid blind search each design step must be clearly motivated, which in practice requires organization of the transformation sequence according to high-level plans. An outline of the plans and transformations we have developed is given in Section 3, followed in Section 4 by the sample derivation of a shortest path algorithm. Sections 5 and 6 discuss extensions and conclude.

This methodology is intended for eventual implementation within the CHI program synthesis system [3], which is under development at Systems Control Inc.

§2 Process graph representation of algorithms

Our choice of representation is motivated largely by our concentration on the earlier phases of algorithm design, in which global restructurings of the algorithm take place. Most data structure decisions can be safely left for a later phase, so we consider only simple, abstract data types like sets, sequences and relations. More importantly, we observe that conventional high-level languages impose a linear order on computations which is irrelevant to the structure of many algorithms and in other cases forces a premature committment to a particular algorithm. To avoid this problem, we have chosen a communicating process representation in which each process is a node in a directed graph and processes communicate by sending data items along the edges which act as FIFO queues. Cycles are common and correspond to loops in a conventional language.

The use of generators (or producers) in algorithm design was suggested by [4]. Our representation is essentially a specialized version of the language for process networks described in [5]. Rather than strive for a general programming language we use only a small set of process types, chosen so that: (1) the specifications and algorithms we wish to deal with are compactly represented, and (2) plans and transformations can be expressed in terms of adding, deleting or moving process nodes. The four process types are:

Generator: produces elements one by one on its output edge.

Constraint: acts as a filter; elements that satisfy the constraint pass through.

Mapping: takes each input element and produces some function of it. If the function value is a set its elements are produced one by one.

Ordering: permutes its input elements and produces them in the specified order.

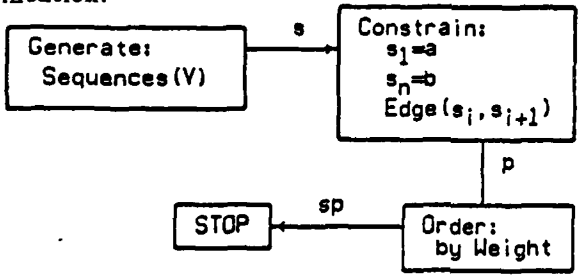
The representation is recursive, a very important property. There can be generators of constraints, constraints on constraints, mappings producing generators, etc. Most of the same design methods will apply to these "meta-processes".

To illustrate the representation, we encode the specification for our sample problem of finding the shortest path from a to b in a graph.

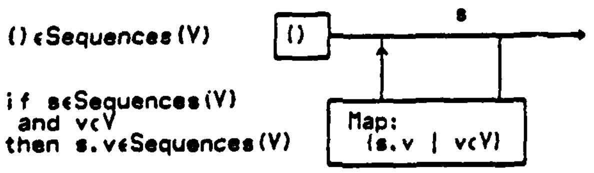
2.1 Notation and terminology for shortest path

A directed graph is defined by a finite vertex set V and a binary relation Edge(u, v). A path p is a sequence of vertices $(p_1 \ldots p_n)$, in which Edge(p_i, p_{i+1}) holds for each pair. The "." operator is used to construct sequences: $(u \ldots v).w = (u \ldots vw)$. Every edge of the graph is labelled with a positive weight $W(u, v)$ and the weight of an entire path is then Weight$(p) = W(p_1, p_2) + \ldots + W(p_{n-1}, p_n)$. The shortest path from a to b is just the one that minimizes Weight.

A specification should be as simple as possible to ensure correctness. Shortest path can be simply specified as: generate all paths from a to b, and select the shortest. We express selection of the shortest path in a rather odd way, feeding all the paths into an ordering process whose very first output will cause the algorithm to stop. The point is that by using a full ordering for this comparatively minor task, we can apply all the plans and transformations for orderings. As for the paths from a to b, they are defined as a certain kind of sequence of vertices, so we introduce a generator of all vertex sequences and place constraints after it to eliminate non-paths. This completes the specification.



Selection of an appropriate internal structure for the generator of Sequences(V) is actually part of the design process, but to simplify the example we will take as a default the usual recursive definition of sequences. The recursion in the definition corresponds to a cycle in the process graph.



The generation process starts when the empty sequence () is produced on the s edge. From the s edge it goes to the constraint and also to the mapping, which produces the set of all one-vertex sequences $().v$, for $v \in V$. These are fed back to generate two-vertex sequences, and so on. A mapping cycle like this is a very common kind of generator.

§3 Methods for algorithm design

The program specification from which design starts is typically written as an exhaustive generate-and-test (or generate-and-minimize) process, and bears little resemblance to the algorithm it will become. The design methods all have the goal of incorporating constraints, orderings or mappings into the generator, or else the goal of planning or preparing to do so. To incorporate a constraint means to modify the generator so that it only generates items which already satisfy the constraint; to incorporate an ordering means to modify the generator so it generates elements directly in that order; and to incorporate a mapping f means to generate elements f(x) instead of elements x.

Accordingly, the methods fall into three main classes, briefly described below. Superimposed upon this class division is a heirarchy (not strict) with multi-step plans at the higher levels and a large number of specific syntactic transformations at the bottom. The heirarchy is organized according to goals and subgoals. Heuristics and deduction rules are required to support the planning activity. At the time of writing, a total of about 20 methods have been formulated not counting low-level syntactic transformations.

Constraint methods. The goal of constraint methods is to reduce the number of elements generated. The top level plan for constraints says to:

1. *propagate* constraints through the process graph to bring them adjacent to a generator,
2. *incorporate* constraints into a generator whenever possible, and if the results are not satisfactory,
3. *deduce* new constraints beyond those given in the specification, and repeat.

Each of the three subtasks is nontrivial in itself and is carried out according to a set of (roughly speaking) intermediate-level methods. For (2), an intermediate-level method that we use several times in the Shortest Path derivation is:

The constraint incorporation plan ConstrainComponent. ConstrainComponent applies when a constraint on composite objects x (sets, sequences, not numbers) is reducible to a constraint on a single component c of x, i.e. $P(x) \equiv P'(x_c)$. ConstrainComponent then gives a plan:

1. Inside the generator, find the sub-generator of values for component c. If necessary, manipulate the process graph to isolate this generator. Again, other methods must be called upon.
2. Remove constraint P and add constraint P' to the sub-generator.

Ordering methods. Another group of methods is concerned with the deduction, propagation and incorporation of orderings on a generated set. These methods are analogous to the methods for constraints but more complicated. In the Shortest Path derivation we use a powerful transformation, explained rather sketchily here:

The ordering incorporation transformation InterleaveOrder. InterleaveOrder applies when an ordering R is adjacent to a generator consisting of a mapping cycle, in which the mapping f has the property $R(x, f(x))$ for all x. In other words, $f(x)$ is greater than x under the ordering R. InterleaveOrder moves the ordering inside the mapping cycle and adds a synchronization signal to make the ordering and mapping operate as coroutines. The ordering produces an element x, the mapping receives it and produces its successors $f(x)$ (there would be no need for the ordering at all if f were single-valued), then the ordering produces the next element and so on.

Mapping methods. The methods for incorporating a mapping into a generator are *mostly based upon recent work in the "formal differentiation of algorithms"* [6] and are related to the well-known optimizing technique of reduction in operator strength. (They are not used in our sample design.)

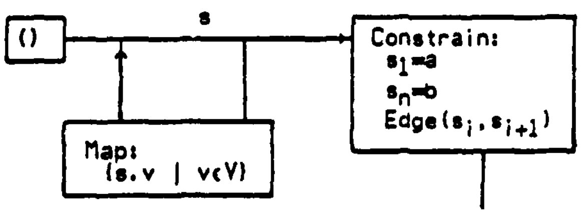
Some syntactic transformations and other methods not described in this section will appear in the derivation.

§4 Example: Design of a shortest path algorithm

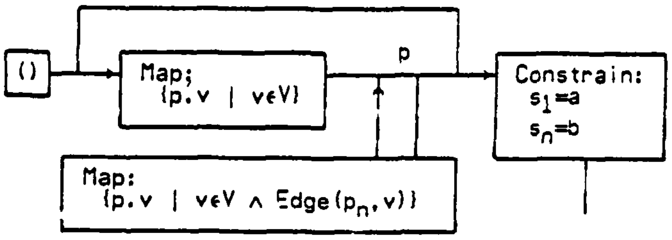
In the design which follows, the specification will be transformed from an inefficient generate-and-minimize scheme into a dynamic programming algorithm. The final algorithm grows paths out from vertex a, extending only the shortest path to each intermediate vertex, until reaching b. Of necessity we omit many details of the design.

4.1 Constraint methods

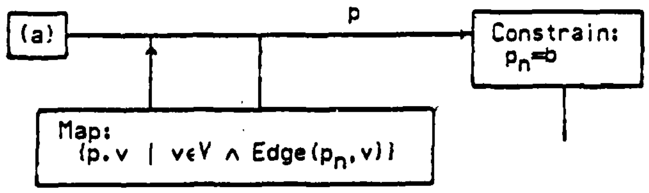
Since the specification's constraints are already next to the generator (step 1), the overall plan for constraints says to try to incorporate them (step 2.) We will follow the heuristic of incorporating the strongest constraint first. Right now, the algorithm reads



Incorporate the Edge constraint. More detail will be shown in this first step than in later derivation steps. ConstrainComponent applies because once a vertex s_i has been added to a sequence, the constraint Edge(s_i, s_{i+1}) reduces to a constraint on the single component s_{i+1}. (This reasoning step is really the application of another method, not described here.) Step (1) in the ConstrainComponent plan says to find the generator of values for components s_{i+1}. Though we have written it in linear form for convenience, the expression $\{s.v \mid v \in V\}$ is really a generator followed by a mapping. Unfortunately "$v \in V$" generates s_1 as well as the desired s_{i+1} values, so we have to unroll one cycle of the graph to isolate the generator of $si + 1$ values. (Again, we have applied methods not described in this paper.) Step (2) is now possible and consists in constraining v to satisfy Edge(s_n, v). With the Edge constraint incorporated, only paths are now being generated so we change s to p in the diagram.



Incorporate the constraint that $p_1 a$. Since the $p_1 = a$ constraint refers only to a component of p, ConstrainComponent applies again. We constrain v in the first "$v \in V$" generator to be equal to a. After simplifying, we obtain



Incorporate the constraint that $p_n = b$. Once again ConstrainComponent applies. This time, however, we are unable to isolate a generator for the last vertex of paths. The last vertex of one path is the next-to-last vertex of another, and so on. ConstrainComponent fails, other methods fail too; we leave the $p_n = b$ constraint unincorporated.

Deduce new constraint. In accordance with the general constraint plan (step 3) we now try to deduce more constraints. One method for deducing new constraints asks: do certain of the generated elements have *no effect whatsoever* upon the result of the algorithm? If the answer is "yes", try to find a predicate that is false on the useless elements, true on others. Motive: if we later succeed in incorporating this constraint into the generator, the useless elements will never be produced.

Now consider the Order + STOP combination. Because all it does is select the shortest path, any path which is *not* shortest will have no effect! The corresponding constraint says:

p is a shortest path from a to b

A further deduction gives the even stronger constraint that *every subpath* of p must be a shortest path (between its endpoints). Incorporation of this constraint is complex and is deferred till after incorporation of the Weight ordering.

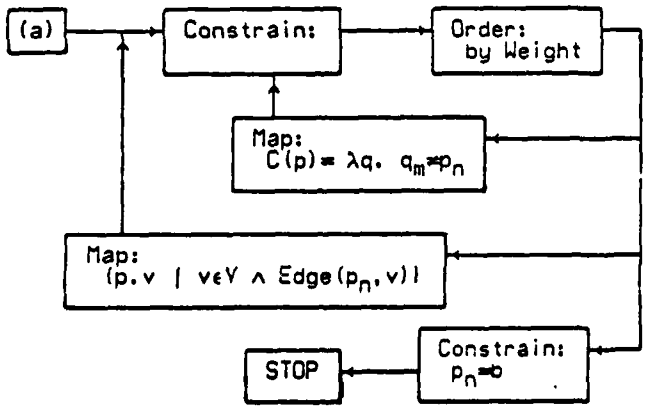
4.2 Ordering methods

So far paths are generated according to the partial order of path inclusion; path p is generated before path q if $q = p.u\ldots v$ for some vertices $u, \ldots, v$. We may generate a lot of paths to b before generating the shortest one - possibly an infinite number. However if the Weight ordering can be incorporated into the path generator, then only a single path to b (the shortest one) will ever be generated.

Propagate Ordering. Before applying an incorporation method we need to bring the Weight ordering next to the generator. Constraints and orderings commute so this is easy.

Incorporate the ordering into the generator. The InterleaveOrder method applies, because Weight($p.v$) is greater than Weight(p). It moves the ordering from outside the generator cycle to inside and also causes the ordering to wait for the mapping to finish extending the previous path before it produces another.

Incorporate new constraint. The "p is a shortest path" constraint is readily incorporated now: the shortest path to any vertex will be the *first* path to that vertex. Any later path q, with the same last vertex $q_m = p_n$, can be eliminated by a new constraint $C(p) = \lambda q.q_m = p_n$. We introduce a mapping to produce these new constraints $C(p)$, and now we have a *generator* of *constraints*. The result of the last three steps is



The algorithm is now a breadth-first search for a path to b, with elimination of non-optimal paths at every vertex. Despite various inefficiencies that remain, the essential structure of a dynamic programming algorithm is present. One interesting improvement comes from incorporating the generated constraints $C(p)$ into the generator of paths, using ConstrainComponent. To complete the derivation would require data structure selection and finally a translation into some conventional programming language.

§5 Other results and limitations

Besides the Shortest Path algorithm shown here (and variants of it) the algorithm design methods have been used to derive a simple maximum finding algorithm and several variants on prime finding including the Sieve of Eratosthenes and a more sophisticated linear-time algorithm. In these additional derivations, no new process types and only a few new methods had to be used. Single and multiple processor implementations have informally been obtained from process graph algorithms, for both prime finding and Shortest Path.

More algorithms need to be tried before specific claims about generality can be made. The intended domain of application is combinatorial algorithms, especially those naturally specified as an exhaustive search (possibly over an infinite space) for objects meeting some stated criteria, which can include being minimal with respect to a defined ordering. Backtrack algorithms, sieves, many graph algorithms and others are of this kind.

The methods described here are quite narrow in the sense that a practical automatic programming system would have to combine them with knowledge of:

1. Standard generators for different kinds of objects. Our methods can only modify an existing generator, not design one.
2. Data structure selection and basic operations such as searching a list.
3. Efficiency analysis to determine if an incorporation really gives a speedup.
4. Domain specific facts, e.g., about divisibility if designing a prime finding algorithm.
5. How to carry out the final mapping of process graph into a conventional programming (or multiprogramming) language.

§6 Discussion and Conclusions

The main lesson to be learned from this work is the importance of using an abstract and modular representation of programs during algorithm design. Details of data structure, low-level operations and computation sequencing should be avoided, if possible, until the basic algorithm has been obtained. (Since some algorithms depend crucially upon a well-chosen data structure, this will not always be possible.) Further, it is advantageous to represent algorithms in terms of a small number of standard kinds of process, for which a relatively large number of design methods will exist. The results so far indicate that just four standard processes suffice to encode a moderate range of different specifications and algorithms. Presumably others will be required as the range is extended, and it is an important question whether (or how long) the number can be kept small. A similar question can be asked about the design methods.

One would not expect methods based upon such general constructs as generators, constraints, orderings and mappings to have much power for the derivation of worthwhile algorithms. For instance, if we had explicitly invoked the idea of dynamic programming, our derivation of a shortest path algorithm would have been shorter. For really difficult algorithms, the general methods may be of little use by themselves. We suggest that they should still serve as a useful complement to more specific methods, by finding speedups (based on incorporation of whatever constraints, orderings and mappings may be present) in an algorithm obtained by the specific methods.

As a final issue, it is interesting to speculate how the stepwise refinement approach to programming might be used by human programmers. Use of a standard set of process types and correctness-preserving transformations would be analogous to the formal manipulations one performs in solving integrals or other equations. If that were too restrictive, perhaps one could use the methods as a guide, without attempting to maintain strict correctness. After obtaining a good algorithm, one could review and complete the design, checking correctness of each transformation step. The result would be a formally correct but also well-motivated derivation.

§7 References

[1] Balzer, Robert; Goldman, Neil; and Wile, David. "On the transformational implementation approach to programming", *Proc. 2nd Int'l Conference on Software Engineering* (1976) 337-344.

[2] Barstow, David R. *Knowledge Based Program Construction*, Elsevier North-Holland, New York, 1979.

[3] Phillips, Jorge and Green, Cordell. "Towards Self-Described Programming Environments", Technical Report, Computer Science Dept., Systems Control, Inc., Palo Alto, California, April 1980.

[4] Green, Cordell and Barstow, David R. "On Program Synthesis Knowledge", *Artificial Intelligence*, 10:3 (1978) 241-279.

[5] Kahn, Gilles and MacQueen, David B. "Coroutines and Networks of Parallel Processes", *Information Processing 77*, IFIP, North-Holland Publishing Company, Amsterdam, (1979) 993-998.

[6] Paige, Robert. "Expression Continuity and the Formal Differentiation of Algorithms", Courant Computer Science Report #5, (1979) 269-658.

[7] Reingold, Edward M., Nievergelt, Jurg, and Deo, Narsingh. *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall Inc., Englewood Cliffs, New

Jersey, 1977.

[8] Elschlager, Robert and Phillips, Jorge. "Automatic Programming" (a section of the *Handbook of Artificial Intelligence*, edited by Avron Barr and Edward A. Feigenbaum), Stanford Computer Science Dept., Technical Report 758, 1979.

[9] Floyd, R. W. "The Paradigms of Programming" (Turing Award Lecture), *CACM* 22:8 (1979) 455-460.

Section VIII

Implementation of Even Squares Derivation

Bernard Mont-Reynaud
Thomas Pressburger

Algorithm specification is facilitated by the use of high level description languages. However, a perspicuous specification at a high level may not immediately suggest an efficient implementation. The work here explores some principles of algorithm design (see section 7) that structure the transformational derivation of an efficient implementation from a high level specification.

The claim is that the use of *constraint* and *generator* constructs clarifies the specification of algorithms and provides guidance in the derivation of implementations. The transformational derivation process becomes structured around the following operations: 1) *deriving* new constraints; 2) *propagating* the constraints to other places in the program; and 3) *incorporating* constraints into generators. A constraint is *incorporated* into a generator by building a generator that produces only those elements that satisfy the constraint. In terms of the generated set of elements, the computation of a constrained subset of a set S could proceed by generating the elements of S and keeping only those elements that satisfy the constraints. Often, however, a constraint is expensive to test, and it is more efficient to generate just those elements satisfying the constraint directly than filter the larger set by the constraint.

A simple example of constraint incorporation is found in the high level specification of the set of even numbers as a generator of natural numbers with a constraint that the output be even. This suggests a program that first produces natural numbers and then filters out those that don't pass a test of being even. It is clear that the even constraint can be incorporated by modifying the natural number generator to multiply each generated natural number by two. This is a more efficient way to generate even numbers.

This section will give a derivation of a program that computes all even perfect squares less than a given MAX. A good part of the derivation has been implemented and runs on the CHI knowledge-based program synthesis system. This simple problem demonstrates techniques of constraint derivation, propagation, and incorporation. The computer implementation expresses the problem using a notation that reflects the use of constraints and generators. The program is described using two set operations of the V language, which is the specification language used throughout the CHI system. The expression **S when P** denotes the set obtained by selecting only those elements from **S** that satisfy the predicate **P**. Another notation for this is $\{x | x \in S \wedge P(x)\}$. The expression **S map F** collects into a set the values of **F** applied to each member of **S**. Another notation for this is $\{F(x) | x \in S\}$. The set that is to be computed in the even squares problem is the subset of natural numbers that satisfy the three constraints of being even, a perfect square, and less than **MAX**. This desired set is succinctly expressed in V using the **when** construct:

```
'INT when 'SQUAREP and 'EVENP and (lambda (X) X < 'MAX)
```

INT is the set of integers, and **SQUAREP** and **EVENP** are predicates about which the system has other information, as will be shown later. The **when** construct has lower syntactic precedence than **and**. Notice that if implemented naïvely, this "program" would specify an infinite computation, because the set of integers is infinite. We will derive a program that computes the result in time $\sqrt{\text{MAX}}/2$.

There are, as is usual, many implementations for the high level specification of the even squares program, and many derivations for each of the implementations. The simplest derivation of a program will be given, where the first step is transforming the constraint that the results be perfect squares into a generator of perfect squares.[1] The incorporation is accomplished in the implementation by giving a transformation rule that incorporates a constraint into a generator when the constraint predicate has a certain general form that the perfect square constraint matches. The next step in the derivation is to incorporate the evenness constraint into the generator of perfect squares by using the fact that if the square of x is even, then so is x. This implies that the even perfect squares are squares of even numbers. The even constraint is transformed into a generator of even numbers using a similar derivation to that given for transforming the perfect square constraint into a generator of squares. Finally, the bound constraint is incorporated as an upper bound on the natural number generator.

We give below the major steps of the derivation at level higher than CHI can accept at present, but in the direction we wish to pursue. The **when** and **map** operations have the same syntactic precedence, so that, for example **S when P map F** can be read left to right as first constraining **S** by **P** and then mapping by **F**.

[1] It is slightly more difficult to derive an efficient program if the even constraint is incorporated first. This requires later in the derivation a fact about when an even number is a perfect square which is more clumsy to express than when a perfect square is an even number.

```
'INT when 'SQUAREP and 'EVENP and (lambda (X)X < 'MAX) end

INCORPORATE SQUAREP CONSTRAINT

'NAT map 'SQR when 'EVENP and (lambda (X)X < 'MAX)

PROPAGATE EVENP CONSTRAINT

'NAT when ('EVENP of 'SQR) map 'SQR when (lambda (X)X < 'MAX)

SIMPLIFY

'NAT when 'EVENP map 'SQR when (lambda (X)X < 'MAX)

INCORPORATE EVENP CONSTRAINT

'NAT map 'DBL map 'SQR when (lambda (X)X < 'MAX)

INCORPORATE BOUND CONSTRAINT

'NAT when (lambda (X) X< SQRT('MAX)/2) map 'DBL map 'SQR
```

The only reasoning operation in the present CHI system is the application of a transformation rule. This makes it difficult to express certain inferences and simplifications which occur in the derivation. These limitations are discussed when they surface, and extensions to CHI are discussed in the conclusion.

Demonstration Using CHI

We present below a transcript of a use of CHI to implement a program that calculates those integers which are even, square, and less than MAX. The transcript paragraphs below are in the form:

```
<context-name> <line-number> . <user-command>     ; <comment>
<CHI-response>
```

The **<context-name>** is a name or number that changes with each application of a rule, and could have been used at any point in the program to revert to previous contexts so as to explore other derivation paths. The **<line-number>** is incremented with each command given by the user.

```
TOP 5. PN       ; this command Prints the current expression
program EVENSQUARES
  begin 'INT when ('SQUAREP and 'EVENP) and (lambda (X)X < 'MAX) end
```

Incorporate Squarep Constraint

The squarep constraint will be transformed into a generator of square numbers. The transformation rule that incorporates a constraint into a generator first requires that the constraint be isolated in a **when** operation. The form of the constraint predicate may then suggest a way to generate directly the elements satisfying the constraint. This will turn out to be the case for the SQUAREP predicate.

The SQUAREP constraint will be isolated by using the SPREADWHEN rule, which will be set up by using associativity of AND.

```
TOP 6. F AND           ; Find AND in the program
('SQUAREP and 'EVENP) and (lambda (X)X < 'MAX)
```

ASSOCAND regroups AND expressions, which is a valid rule because AND is associative.

```
TOP 7. PR ASSOCAND  ; PRint the ASSOCAND rule
rule ASSOCAND TRANSFORM / *:((A and B) and C) /
                      => / *:(A and (B and C)) /

TOP 8. A ASSOCAND   ; Apply the ASSOCAND rule
'SQUAREP and ('EVENP and (lambda (X)X < 'MAX))
Task completed
```

We are now ready to apply SPREADWHEN to isolate the SQUAREP constraint.

```
1, 9. 0              ; go up to the expression containing the current one
'INT when 'SQUAREP and ('EVENP and (lambda (X)X < 'MAX))
```

Filtering a set through a conjunction of predicates can be implemented by filtering the set through each of the predicates successively. SPREADWHEN performs this transformation.

```
1, 10. PR SPREADWHEN
rule SPREADWHEN TRANSFORM / *:(S when C1 and C2) /
  => / *:(S when C1 when C2) /

1, 11. A SPREADWHEN
'INT when 'SQUAREP when 'EVENP and (lambda (X)X < 'MAX)
Task completed
```

Notice that in general one would have to apply commutativity and associativity rules several times to place the desired conjunct in the first position where it could be extracted by SPREADWHEN. This is detail that the user should not be required to perform. An inference system and a higher-level plan would at least allow a command like "isolate the **SQUAREP** constraint", which is a step towards a convenient interface.

We will now look at the definition of SQUAREP so as to be able to generate squares directly, rather than generating integers and testing each for satisfying SQUAREP.

```
2, 12. F SQUAREP
'SQUAREP

2, 13. PR SQUAREPDEF
rule SQUAREPDEF TRANSFORM / *:'SQUAREP /
  => / *:(lambda (V)exists Y in 'NAT st V=(sqr Y)) /  & V:'BINDING
      & Y:'BINDING & V.'NAME=(GENSYM 'V) & Y.'NAME=(GENSYM 'Y)
```

SQUAREP(x) tests to see if there is a natural number y whose square is x.

```
        2, 14. A SQUAREPDEF
        (lambda (V1)exists Y1 in 'NAT st V1=(sqr Y1))
        Task completed
```

The key to finding the generator for the SQUAREP constraint is that the form of the SQUAREP predicate fits a certain *existential form* which generally allows reformulating the calculation of the set of integers filtered by SQUAREP as "sqr"ing natural numbers. We focus on the existential statement, to transform it into a predicate reflecting the above fact.

```
        3, 15. F EXISTS
        exists Y1 in 'NAT st V1=(sqr Y1)
```

The next rule, **FINDMAPPEDSET** expresses the semantics of what it means for an element to be in the value of a set mapped by a function. The rule is slightly wrong: it should test that **X** is not a free variable of the expression **V**, but is a free variable in expression **F**.

```
3, 16. PR FINDMAPPEDSET
rule FINDMAPPEDSET TRANSFORM / *:exists X in SET1 st V=F / & V:'BINDING
  => / *:(V in (SET1 map (lambda (X)F))) /

3, 17. A FINDMAPPEDSET
V1 in ('NAT map (lambda (Y1)(sqr Y1)))
Task completed
```

What has happened is that the existential form of the **SQUAREP** predicate is an alias for expressing the constraint as membership in the set of squares. The redundant lambda will be removed by rule **REMLAMBDA**. This should become a simplification rule that is invoked automatically.

```
    4, 18. F LAMBDA
    (lambda (Y1)(sqr Y1))
```

This next rule accomplishes the equivalence: (lambda (x) F(x)) <=> 'F

```
4, 19. PR REMLAMBDA
rule REMLAMBDA TRANSFORM
  *:'FUNCTION & *.'STEPS=B & *.'ARGS=A & B:C & (DUMBLAMBDAP * C)
  => *:'CONSTANT & *.'NAME=C

4, 20. A REMLAMBDA
'SQR
Task completed

5, 21. 0 0 0 0      ; ascend to the containing when expression.
'INT when (lambda (V1)V1 in ('NAT map 'SQR))
```

The result of this last expression is the set whose elements are in both sets 'INT and 'NAT map 'SQR. The rule below expresses this fact in general. The INTERSECT operation can then be simplified.

```
5, 22. PR WHENTOINTERSECT
rule WHENTOINTERSECT TRANSFORM / *:(SET1 when (lambda (X)X in SET2)) /
  => / *:INTERSECT (SET1 SET2) /

5, 23. A WHENTOINTERSECT
INTERSECT ('INT 'NAT map 'SQR)
Task completed
```

The set of squared numbers is certainly a subset of the set of integers. The current system cannot use this fact to simplify the INTERSECT operation. We comment on a possible extension of the current system in the concluding remarks of this section. Currently, this expression is simplified using the following rule.

```
6, 24. PR SIMPLIFY-SQRNAT
rule SIMPLIFY-SQRNAT TRANSFORM / *:INTERSECT ('INT 'NAT map 'SQR) /
  => / *:('NAT map 'SQR) /

6, 25. A SIMPLIFY-SQRNAT
'NAT map 'SQR
Task completed
```

Propagate Evenp Constraint

The remaining constraints of the original program need to be incorporated. We'll work on the evenness constraint next.

```
7, 26. 0
'NAT map 'SQR when 'EVENP and (lambda (X)X < 'MAX)
```

The EVENP constraint is first isolated.

```
7, 27. A SPREADWHEN
'NAT map 'SQR when 'EVENP when (lambda (X)X < 'MAX)
Task completed

8, 28. 1
'NAT map 'SQR when 'EVENP
```

The evenness constraint is more difficult to incorporate directly because it appears after a **map** operator. The trick is to move the EVENP constraint past the SQR mapper. This movement can always be performed (using SWMAPWHEN) when dealing with contraints of a single variable. If the expression is then simplifiable, a cost reduction is achieved, because then the **map** operation is performed only on those elements passing the simplified constraint, rather than on every element. In our case, we have even better luck: the simplified constraint can be incorporated into a generator.

```
8, 29. PR SWMAPWHEN
rule SWMAPWHEN TRANSFORM /*:((S map F) when C)/
  => /*:(S when (C of F) map F)/

8, 29. A SWMAPWHEN
'NAT when ('EVENP of 'SQR) map 'SQR
Task completed
```

The **when** predicate can be simplified.

```
9, 30. F OF
'EVENP of 'SQR
```

Simplify

The following rule uses the fact that the square of a number is even if and only if that number is in fact even. This is perhaps the crux of this derivation: even squares are in fact squares of evens, and it is easy to square numbers and generate evens. At present, CHI is unable to prove this rule from a more basic set of facts about divisibility.

```
9, 31. PR EVENPSQR
rule EVENPSQR TRANSFORM / *:'EVENP of 'SQR /  => / *:'EVENP /

9, 32. A EVENPSQR
'EVENP
Task completed
```

Incorporate Evenp Constraint

Now, we explore the definition of **EVENP** to find a generator using a derivation similar to the one used when we changed **SQUAREP** into a generator. The first rule says that an even number is one which is the double of a natural number.

```
10, 33. PR EVENPDEF
rule EVENPDEF TRANSFORM / *:'EVENP /
  => / *:(lambda (V)exists Y in 'NAT st V=(dbl Y)) /  & V:'BINDING
       & Y:'BINDING & V.'NAME=(GENSYM 'V) & Y.'NAME=(GENSYM 'Y)

10, 34. A EVENPDEF
(lambda (V2)exists Y2 in 'NAT st V2=(dbl Y2))
Task completed
```

We express the existential as membership in a mapped set.

```
     11, 35. -1
     exists Y2 in 'NAT st V2=(dbl Y2)

     11, 36. A FINDMAPPEDSET
     V2 in ('NAT map (lambda (Y2)(dbl Y2)))
     Task completed
```

The redundant lambda is removed...

```
     12, 37. F DBL
     (dbl Y2)

     12, 38. 0
     (lambda (Y2)(dbl Y2))
     12, 39. A REMLAMBDA
     'DBL
     Task completed

     13, 40. 0 0 0 0
     'NAT when (lambda (V2)V2 in ('NAT map 'DBL))
```

The last expression is rewritten as an **INTERSECT** operation.

```
13, 41. A WHENTOINTERSECT
INTERSECT ('NAT 'NAT map 'DBL)
Task completed
```

A simplification rule is used, in place of reasoning about doubling.

```
14, 42. PR SIMPLIFY-DBLNAT
rule SIMPLIFY-DBLNAT TRANSFORM / *:INTERSECT ('NAT 'NAT map 'DBL) /
  => / *:('NAT map 'DBL) /

14, 43. A SIMPLIFY-DBLNAT
'NAT map 'DBL
Task completed

15, 44. 0 0
'NAT map 'DBL map 'SQR when (lambda (X)X < 'MAX)
```

The last constraint needs to be percolated down into a generator of natural numbers (e.g. **for i from 1 to SQRT(MAX)/2**). Reasoning about the inverses of **SQR** and **DBL** needs to be used, some of which CHI is unequipped to handle. So the implementation of the derivation will stop here for now. We will give a summary extended derivation, which has not yet been implemented.

```
'NAT map 'DBL map 'SQR when (lambda (X)X < 'MAX)
Apply SWMAPWHEN
'NAT map 'DBL when (lambda (X)X < 'MAX) of 'SQR map 'SQR
Simplify composition
'NAT map 'DBL when (lambda (x) sqr(x) < 'MAX) map 'SQR
Prove that x is in 'NAT, then use inverse of SQR:
'NAT map 'DBL when (lambda (x) x<sqrt('MAX)) map 'SQR
Apply SWMAPWHEN
'NAT when (lambda (x) x<sqrt('MAX)) of 'DBL map 'DBL map 'SQR
Simplify composition
'NAT when (lambda (x) dbl(x)<sqrt('MAX)) map 'DBL map 'SQR
Use inverse of dbl:
'NAT when (lambda (x) x<half(sqrt('MAX))) map 'DBL map 'SQR
```

Summary and Remarks

The following rules were used in the derivation and are of general use in programs that contain set mapping and constraining expressions, functions or conjunctions.

```
rule ASSOCAND TRANSFORM / *:((A and B) and C) /
                        => / *:(A and (B and C)) /

rule SPREADWHEN TRANSFORM / *:(S when C1 and C2) /
  => / *:(S when C1 when C2) /

rule FINDMAPPEDSET TRANSFORM / *:exists X in SET1 st V=F / & V:'BINDING
  => / *:(V in (SET1 map (lambda (X)F))) /

rule REMLAMBDA TRANSFORM
  *:'FUNCTION & *.'STEPS=B & *.'ARGS=A & B:C & (DUMBLAMBDAP * C)
  => *:'CONSTANT & *.'NAME=C

rule WHENTOINTERSECT TRANSFORM / *:(SET1 when (lambda (X)X in SET2)) /
  => / *:INTERSECT (SET1 SET2) /

rule SWMAPWHEN TRANSFORM /*:((S map F) when C)/
  => /*:(S when (C of F) map F)/
```

The following rules present definitions of perfect squares and even numbers.

```
rule SQUAREPDEF TRANSFORM / *:'SQUAREP /
 => / *:(lambda (V)exists Y in 'NAT st V=(sqr Y)) /  & V:'BINDING
      & Y:'BINDING & V.'NAME=(GENSYM 'V) & Y.'NAME=(GENSYM 'Y)

rule EVENPDEF TRANSFORM / *:'EVENP /
 => / *:(lambda (V)exists Y in 'NAT st V=(dbl Y)) /  & V:'BINDING
       & Y:'BINDING & V.'NAME=(GENSYM 'V) & Y.'NAME=(GENSYM 'Y)
```

These rules present transformations that could be derived from facts about **'DBL** and **'SQR**.

```
rule EVENPSQR TRANSFORM / *:'EVENP of 'SQR /  => / *:'EVENP /

rule SIMPLIFY-SQRNAT TRANSFORM / *:INTERSECT ('INT 'NAT map 'SQR) /
  => / *:('NAT map 'SQR) /

rule SIMPLIFY-DBLNAT TRANSFORM / *:INTERSECT ('NAT 'NAT map 'DBL) /
  => / *:('NAT map 'DBL) /
```

The latter two rules should be expressed differently, perhaps using facts deduced about the program.

```
rule SimplifyIntersection /*: INTERSECT(S T)/ & SUBSETP(S,T) => /*:S/

Fact RangeSqr ForAll x . x in 'INT => sqr(x) in 'NAT

Fact RangeDbl ForAll x . x in 'NAT => dbl(x) in 'NAT

Fact NatInt  'NAT subset 'INT
```

Appropriate use of the **Facts** above will allow deducing **SUBSETP('NAT map 'SQR, 'INT)**, so that **INTERSECT('NAT map 'SQR, 'INT)** will simplify to **'NAT map 'SQR**. In fact, all the transformation rules could become the substitution of equals for equals if we give facts like:

```
Fact EVENPDEF ForAll x . EVENP(x) = exists y in 'NAT st x = dbl(y)
```

We suggest developing CHI in the directions presented, so as to make it more adequate in dealing with reasoning about programs.

Acknowledgements: Cordell Green and Steve Tappel made helpful comments on the content and form of this paper.

AD-A105 661   SYSTEMS CONTROL INC PALO ALTO CA COMPUTER SCIENCE DEPT   F/G 9/2
RESEARCH ON KNOWLEDGE BASED PROGRAMMING AND ALGORITHM DESIGN.(U)
AUG 81 C GREEN.   N00014-79-C-0127
UNCLASSIFIED SCI-ICS-L-81-5   NL

2 OF 2
AD
A105661

END
DATE
FILMED
11-81
DTIC

Section IX

Divide and Conquer Algorithms

Jorge Phillips

This section illustrates the application of the divide and conquer design plan.

Let us assume that the user wants to develop an efficient implementation of searching for the occurrence of an element in a set, i.e. testing for membership. The most straightforward specification for this is to define (name) a function whose purpose is to perform the test. Thus, the user types in

$$search \Leftrightarrow x \in S$$

which defines the name *search* denoting the test.

The first step is to parse this string and translate it into internal structure. The internal structure is a representation for the specification, which can be manipulated and annotated easily by the system. It corresponds to what is generally termed the "abstract syntax" of the string, a structure which reflects the semantic relations between the abstract components of the piece of description. This representation takes the form of a network of relations between objects and their descriptions. A description can be viewed as a repository of all properties, structural and deductive information about a set of objects which are instances of the description, i.e. members of the class. The network is composed of nodes which denote program components (like a variable or an operation) and arcs linking the nodes, which denote relations between them (for example the relationship between an application of an operation and its argument variables). In turn, every component of the graph is linked to generic class descriptions (also called generic types) for the class that the particular node belongs to. These descriptions are used to annotate the abstract syntax for the specification and constrain its components. For example, in the specification input by the user there are several primitive operations. One of them is $\in$, a membership test, and the other is $\Leftrightarrow$, a naming operation. Thus, for example the description of $\in$ is:

This appendix has been extracted from a draft of Jorge Phillips' forthcoming Ph. D. thesis

```
description ∈
a PrimitiveOperation
  with  arguments
            (a data-object X)
            (a data-object Y)
        constraints
            set(Y)
            compatible (type(X)
                        type(prototype-element(Y)))
        produces  (a boolean)
        ... other assertions and properties ...
```

From the descriptions for the components of the abstract syntax for *search* it is inferred that it is naming a boolean-valued operator (a predicate), since it is a naming a membership test, an operation already known to belong to this class. *Search* is annotated as needing two arguments, which will be aliases to the arguments of the membership operation. The surface annotations have thus transformed the internal structure into

```
function search (arg1, arg2) : boolean ⇔ x ∈ S

is-alias(arg1,x)
is-alias(arg2,S)
```

Descriptions carry with them constraint information that propagates to all their instances. In particular, in this example constraint information for ∈ says that it must be the case that the type of x be the same as the type of the prototype element of S, and that S must be a set. Thus, the internal abstract syntax is transformed into

```
function search (arg1, arg2) : boolean ⇔ x ∈ S

is-alias(arg1,x)
is-alias(arg2,S)
type(S,set)
constraint type(X)==type(prototype-element(S))
```

§1 Design Methods are Plans

Once the specification has been input the user can choose a design method (i.e. a plan) by which to transform this specification into a program with particular interface characteristics. Usually a design system has several possible design methods ranging from standard programming technology like data structure representation, to sophisticated methods like dynamic programming or path compression, through intermediate methods like divide and conquer. The system may also provide design methods which are based on the concepts of incoporation of constraints to generators.

Design methods are like standard plans: they have a set of preconditions which denote the circumstances in which the method is applicable, and have a set of actions or goals to be carried out through which the top goal, in this case the design of an algorithm that satisfies a specification, is achieved. A design method thus constrains the space of possibilities for the target algorithm by constraining the search space of possible programs to those that can possibly satisfy the specification. Notice here that the applicability of a design method is not only constrained by the abstract syntax of the algorithm specification, but also by the environment the target algorithm is going to run on.[1] In a sense, as we shall see later in this thesis, the evolution of a design is the result of the interplay between an environment and a set of design rules, the design rules generating the space of possible next states for the design and the environment constraining which rules are applicable to generate the space.

Back to design methods, the next step is to choose what global strategy (if this is the path chosen by the user) or plan to use in transforming the specification. The system comes back to the user with a list of possible design methods:

Dynamic Programming
Generate and Test
Path Compression
Incremental Generation
Store vs Recompute
Divide and Conquer
Language Specific
Data Structure Refinement
... etc ...

Lets assume the user requests that the divide and conquer method be applied to the specification. The system searches the descriptions for methods and retrieves it. Here is what the method looks like:

[1]Perhaps there is only one efficient possible data structure implementation for a critical object in the algorithm which precludes from the start the applicability of a particular design method.

```
design method DIVIDE-AND-CONQUER
'f(x1,....,xm,S)' & composite(S) & uniform(S)
                  & (predicate(f) ∨ constructor(f))
→
'if P(x1,....,xm,S)
    then G(x1,....,xm,S)
  else let S1,S2 — split(S)
         do join(f(x1,....,xm,S1),
                 f(x1,....,xm,S2)); '

constrain  partition(split) & predicate(P)
                            & constructor(join)

create-goal('refine split',1)
create-goal('refine join',2)
create-goal('refine P',3)
create-goal('refine G',4)
```

Notice that the method is really a rule for setting up a schema or plan to achieve a procedural specification of the algorithm. A method consists of three parts. First, the preconditions of the plan, namely a "specification pattern" that the method applies to and constraints that must be satisfied by the pattern and its embedding specification. Second, "a result pattern" or schema, which reflects a plan to achieve the algorithm. Finally, a set of "consequent actions" to be performed after instantiation of the plan. Thus, in the above case, the preconditions are the pattern of function invocation (delimited by '...') and the constraints on parts of the pattern, which in this case say that the pattern applies only to functions that compute predicates or construct objects and which are in some way "uniform", i.e. that they exhibit a continuity in the construction or testing process. This constraint is needed, since the divide and conquer design technique requires that the function being computed on an object may be computed from recursive invocation on subparts of the object. The result pattern is the two-dimensional divide and conquer schema. The consequent actions are the constraints on the different parts of the plan and the actions to be effected on the system's agenda.

After having retrieved the method, it is applied. The specification pattern is tested against $search \Leftrightarrow (x \in S)$ with f bound to $\in$. The arguments of a function are considered to be part of a set, i.e. with no explicit ordering, it needs to determine how to bind the $x \leftarrow i$ and S. The default rule for this is to choose the natural ordering (left to right) if the user does not specify a preferred order. Whatever the case is, system and user decide that the pattern instantiates with x as arg_1 and S as arg_2. Since S is a set, the descriptions for sets in the system are searched and the descriptions for the *uniform* and *composite* properties, to determine whether they hold for S. Sets are by nature uniform. This is determined by asking the *uniform* property to define how to compute itself for a *set*. The property's description will then assert that sets are uniform objects, and add this fact to the description of a set, if it had not been added earlier. This will make the precondition succeed.

The preconditions succeed and the system invokes the description base to replace the specification for the corresponding specification in the method's consequent. Of course, all these transformations on descriptions are undoable, in case the elaboration of a plan does not succeed. The final step is to execute the relevant actions in the consequent of the method. In this case, invoking the agenda to insert some goals and constraining the split and join components of the plan to be a partition and a constructor. A more subtle constraint that will require more deductive capabilities is the satisfaction of uniformity of the split and join, i.e. guarantee that the split be an invertible operation. For now, this will be left to the user. A more sophisticated system will incorporate an axiomatization of the domain that the design method is applied in, and deduce the existence of the split-join pair.

§2 Finding the Split

At this point the system has applied the method and replaced the denotation for *search* by

```
function search (x, S) : boolean ⇔
  if P(x,S) then G(x,S)
  else let S1,S2 — SPLIT(S)
          do JOIN( search(x,S1), search(x,S2) );

constraint partition(SPLIT) & predicate(P)
                              & constructor(JOIN)
constraint type(S)=set
constraint type(X)=type(prototype-element(S))

State of agenda:
  1. SPLIT
  2. JOIN
  3. Recursion Termination Predicate P
  4. Base Value Constructor G
```

where the aliases have been absorbed by replacing them with the corresponding variables defined by the user. Notice that the agenda has an ordered set of goals introduced by the method. The first goal is taken from the agenda and tries to elaborate it. The constraint is that *SPLIT* is a partition applied to an object *S* which has been constrained to be a set. Let us assume that the user has decided to constrain the set argument of *search* to be ordered. The user types

S : 'ordered set of integer'

so the system searches for a referent for S and finds it in the current context (i.e. the search function). A new object in the abstract syntax is generated to denote the ordered set, and the old *S* is replaced. The system finds all active constraints for S, and tests that they are maintained invariant, which is the case since S is still a set. Thus the replacement is carried out. In the process of doing this, the constraint is noticed on x which was dependent on the type of the prototype element of S being defined. *Immediately it defines x to have a type of integer.* The constraint on x was represented as a relationship to be maintained invariant between x and S and hanging from the prototype element structure of S. When the S object was created, the constraint got propagated to x. The algorithm was transformed into

```
function search  (x : integer,
                  S : ordered set of integer) : boolean
    ⇔
    if P(x,S) then ...
```

At this point, the system starts working on the SPLIT goal in the agenda. It knows that the splitee is a set, so it searches the description base for splits that can be applied to sets The base includes several kinds of splits all of them standard programming techniques:

Singleton Split
Equal Size Split
Interleaved Generator
...

The first one splits a set into a singleton set and into the rest of the original set after removing the singleton. This is one of the standard splits. The second one splits a set into pieces of equal size. It thus introduces a constraint on the split. The third kind of split generates the set into different subsets. Whichever technique is applied here will have to satisfy the pending constraint that the split operation be a partition. And of course there is variety of other ways in which to split a set. The point here is that the system is confronted with a decision point in the search space. In order to prune the size of the search space, the system includes metarules and constraints on metalevel objects to provide heuristic guidance.

"When selecting a way for splitting a set, try to use first a balanced (equal size) split"

"A 2-split is balanced if the following constraint holds: size($S1$) − size($S2$) ≤ 1, where $S1$ and $S2$ are the components of the split."

The rule is applied which selects the equal size split method. The problem has thus been reduced to finding an equal size split for a set. The following rule accomplishes this:

"An equal size split can be done on an ordered set by splitting the set around its median"

This is clear for then the split is the most balanced split that is obtainable for the set. Notice that the effectiveness of this split is related to the ease with which the median can be computed. The rule for achieving this is:

"Compute the median of the set. For a 2-split create two subsets: Those elements less than the median, and those greater than the median"

This translates into a rule of the form

```
'let S1,S2 —  SPLIT(S) do ...' & S.ordered
                              & equal-size(SPLIT)
→
'let m —  median(S) do
     let S1 —  { y ∈ S ∩  y≤m }
         S2 —  { y ∈ S ∩ y>m } do ...'
```

which transforms the state of the program description accordingly. At this point the split goal has been reduced to the subgoals of finding efficient ways for computing the median of an ordered set and of splitting an ordered set around a pivot element. The agenda thus looks as follows:

```
1. Split
            Median
            Subsetting
2. Join
3. Recursion Termination Predicate P
4. Base Value Constructor G
```

and the system is ready to work on the next top level goal, the *join*.

§3 Finding a Join operation

In the preceding discussion on divide and conquer we saw how the split and the join are in a sense interrelated. We saw also how if a predicate was a "localizable" function then we could find a split-join pair of functions for it. The system has programming knowledge about join operations for predicates. The following two pieces of knowledge are relevant here:

"A Join operation for a split predicate has to be a boolean function. As such it has to be composed of $\wedge$, $\vee$ or $\oplus$ elementary boolean functions"

This rule just states a fact: if we are computing a predicate by recursively splitting the computation then to put back the partial results we need a boolean composition of them. A blind search for a join operation for a predicate will thus generate the possible boolean combinations of the recursive invocations. Usually the search space can be constrained by other information in the domain. This is the case here. The system observes that $S = S_1 \bigcup S_2$. The following chain of reasoning holds

$$S = S_1 \bigcup S_2 \tag{1}$$
$$search(x, S) \Leftrightarrow x \in S \tag{2}$$
$$x \in S \Leftrightarrow x \in S_1 \vee x \in S_2 \tag{3}$$
$$search(x, S) \Leftrightarrow search(x, S_1) \vee search(x, S_2) \tag{4}$$
$$join = \vee \tag{5}$$

and the system replaces $\vee$ for its first guess for the join. Notice that this is a correct, albeit inefficient, join operation. In later steps of the transformation process the join operation will be transformed into a more efficient version.

§4 Finding the Base Case and Termination Condition

There are two top-level goals left to attack, of the four goals introduced by the divide and conquer schema. The next one in the agenda is finding the base case for the recursion. Since a predicate is being computed the following rules hold:

> *"The default termination condition for computing a predicate recursively on a set is a test for a singleton set"*
>
> *"To test whether a set is a singleton set, test whether its size is 1"*

The application of these two rules defines the base case to be $size(S) = 1$. The termination condition then is the computation of $x \in S$ for a singleton set.

> *"If a set is of size 1 then $x \in S$ can be transformed into a test for equality $S = \{x\}$.*

The program at this point looks as follows:

```
function search  (x : integer;
                 S : ordered set of integer) : boolean ⇔
  if size(S)=1 then S = {x}
  else  let m —  median(S) do
        let S1 —   { y ∈ S ﬂ y ≤ m }
            S2 —  { y ∈ S ﬂ y>m } do ...
          do search(x,S1) ∨ search(x,S2);

constraint type(S)=set
constraint type(X)=type(prototype-element(S))

1. Split
      Median
      Subsetting
2. Join
      Improve efficiency of ∨
3. Termination Condition
      Compute size of S
4. Base Value Constructor G
      Test for equality
```

and the original declarative specification for the algorithm has been transformed into a functionally equivalent high level procedural specification. The specification is executable in a substrate where there exists an interpreter for the constructs of the specification language. The task of generating a program is then a task of successive transformation of this new specification via correctness preserving transformation rules into a substrate where the algorithm or program can execute. The original schema offered by the design method has been filled in and the original goal replaced by a partial order of accessory goals to be achieved.

§5 Efficiency Issues

The declarative specification for searching a set has been transformed using heuristic knowledge about programming and about design into a very high level procedural specification. The next step is to transform this description level into an executable specification in some substrate. This process will be based on the interplay of efficiency and transformation choices. Efficiency rules will select places to work on in the program description and transformation rules will take the description closer to a description in the substrate.

In the case of the search algorithm, the program has several places where crucial efficiency choices have to be made. For example, in the case of the divide and conquer method it is crucial to implement the split and join as fast as possible.

> *"The split and join loci in a divide and conquer plan are the most important efficiency targets in the plan"*

The system thus flags the corresponding places in the description: the "$\vee$" operation, the computation of the median, and the subsetting operations. Efficient implementations of these require heuristics to select refinement paths, and deductive capabilities to automate partially the reasoning process involved in finding efficient implementations.

§6 Speeding up the Join

The $\vee$ operation is selected as a likely candidate for further refinement. What can be said of the operation?

$$\begin{array}{l}\forall x \in S . x \in S1 \vee x \in S2 \\ S1 \bigcap S2 = \emptyset \\ \neg \exists x \in (S1 \bigcap S2) \rightarrow (x \in S1) \bigoplus (x \in S2)\end{array}$$

So the $\vee$ operation can be transformed into an exclusive or operation, since only one of the two branches can succeed at any one time. One of the properties of $\bigoplus$ is

$$(x \in S1) \bigoplus (x \in S2) \Rightarrow [(x \in S1) \Rightarrow \neg(x \in S2))\&(x \in S2) \Rightarrow \neg(x \in S1))]$$

Thus, if there exists a discriminating predicate P which tells which branch if at all needs to be evaluated then the $\bigoplus$ can be speeded up.

> *"An exclusive-or ($\bigoplus$) operation can be speeded up by finding a discriminating predicate which can be executed fast, which tells which side of the operation to evaluate"*

This is an example of a predictive rule. It speeds up a computation where one of two paths has to be followed by providing a discriminator that tells which of the two paths can possibly succeed.

$$\text{`}X \bigoplus Y\text{'} \Rightarrow \text{`}if\ P(.)\ then\ X(.)\ else\ Y(.)\text{'}$$

and the P predicate has to satisfy the assertion $\forall z . P(z) \equiv X(z) \wedge \neg Y(z)$ which can be condensed to $-z . P(z) \equiv X(z)$ since $X(z) \Rightarrow \neg Y(z)$. In the case of the $\vee$ operation

$$\begin{array}{l} x \in S1 \equiv x \leq m \equiv x \leq median(S) \\ x \in S2 \equiv x > m \equiv x > median(S) \\ x \leq median(S) \equiv \neg(x > median(S)) \end{array}$$

so by letting P be $x \leq m$, then P is a discriminating predicate for the $\bigoplus$ operation, and can be computed in constant time. The join operation can then be transformed into

$$if\ x \leq m\ then\ search(x, S1)\ else\ search(x, S2),$$

a more efficient join which will allow the logarithmic speed-up to the linear version of the algorithm.

§7 Computing the Median of an Ordered Set

Efficiency analysis recommends the next task to work on. Recall that both the split and join foci are important efficiency considerations in the design of the algorithm. Having worked on the join operation, the next part of the program description to be worked upon is the computation of the median of the set. This computation is dependent on the implementation of the set. The problem of implementing the median efficiently is reduced to the problem of representing an ordered set in a way which makes it very efficient to compute it.

§8 Selecting a Representation for an Ordered Set

Associated with each description of a class of objects there is a set of rules which can implement (i.e. realize) objects of the class at lower substrate levels. In the case of a set the space of possible implementations is large (characteristic function, relation, record, list, etc.). Some of these make it more easier to compute the median of the set than others. Choices of which implementation to choose are made either by the user or mediated by heuristics in the system. In this case, the system has the heuristic that says:

"The computation of the median of an ordered set can be done efficiently by representing the ordered set as a mapping"

The idea here is again to reduce the computation of the median of the set, which may in the worst case require a linear order of operations, to a constant order by transforming the set into a mapping from ordinals of the set into elements of the set. Thus, if F is the mapping, and S is the related set, then

median(S) = F(median(1..size(S)))

which can be accomplished in constant time if the set is represented explicitly or if this is not the case, if the size of the set is computable in constant time. The rule that accomplishes this transformation is the following:

```
'ordered set of M' & *.uses=MEDIAN
    ⇒
'mapping from 1..|S| to M'
```

Notice that we also need a metarule that says

> *"If the locus of efficiency is the computation of the median of a set, and the set is ordered, then apply a rule that takes the set into a mapping."*

This metarule will force the refinement path of the set to fall into a mapping. How is it decided how to compute the mapping once the set's representation is chosen? It is done by definition of the mapping

$$\forall(x \in S)\exists(i \in 1..size(S)) \mid F(i) = x$$

In particular, there must be some index which maps into the median. The deduction step, which may be filled in by the user, is realizing that the index for the median element of the set is the median of the set of indices of the domain of the mapping. Once this is done, the implementation for the median computation is:

$$median(S) \Rightarrow S(median(1..size(S)))$$

Now the problem is to determine how to compute the median of a constant set, in this case of a set of ordered contiguous integers. The definition for median in this case is:

$$median(1..n) = \left\lfloor \frac{1+\mathrm{n}}{2} \right\rfloor$$

This definition is replaced and produces the following piece of code:

$$\textbf{let m be S}(\left\lfloor \frac{1+\mathrm{n}}{2} \right\rfloor) \quad \textbf{do} \ldots$$

§9 Propagating the Effects of a Representation Choice

The representation of the ordered set as a mapping entails changes in representation of several operations. There are several places in the program description that need representation changes: the subsetting constructs, the size computation, the equality test, etc. These propagations can be carried out by rules or associated with the representation change.

> *"If a set is represented as a mapping from ordinals into elements of the set, testing for equality of the set against another set transforms into testing for equality of the range of the mapping against it."*

"Finding a subset of a set which is represented as a mapping can be accomplished by finding the mapping of a subset of the ordinals of the set, for some ordinal subset depending on the original subset"

"If a set is represented as a mapping from ordinals into elements then computing the size of the set transforms into computing the size of the domain of the set"

These three rules belong to a class of rules that express the semantics of operations on some data type in terms of operations on another data type[2]. They may be viewed as legal move generators, or constraints, in the space of transformations for a synthesis system. Application of the first rule generates the transformation

$$S = \{x\} \Rightarrow range(S) = \{x\}$$

Application of the second rule needs deduction to infer the domain subset that corresponds to the subset operations in the algorithm specification. The following chain of reasoning enables the necessary inferences to be made.

$$\begin{aligned}
&\{\, x \in S \mid x \leq m \,\} \text{ where } m = median(S) \\
&S^{-1}(m) = median(1..size(S)) \\
&\{\, x \in S \mid x \leq m \,\} = S(\{\, x \in 1..size(S) \mid x < median(1..size(S)) \,\}) \\
&\{\, x \in S \mid x \leq m \,\} = S \mid 1..m' \text{ where } m' = median(1..size(S))
\end{aligned}$$

In this way a subset operation on the original set has been transformed into a restriction of the mapping to a subset of its domain, thanks to the ordering relation on the original set. After application of the third rule which transforms computing the size of the set to computing the size of the domain of the mapping the program description thus becomes:

[2]The approach used is that of expressing data type semantics in terms of the semantics of a kernel of operations and a set of transformations from the data types to data types in the kernel. The kernel acts as a semantic base for the system.

```
function search  (x : integer;
                       S : mapping from 1..size(S)
                                             into integer) : boolean ⇔
if size(S)=1 then range(S)={x}
   else  let m be S(⌊ $\frac{1+n}{2}$ ⌋) do
           let S1 =  S |  {1..m}
                S2 =  S |  {(m + 1)..size(S)}
           do if x≤m
                    then search(x,S1)
                else search(x,S2);
```

1. **Split**
   - **Median**
   - **Restriction**
2. **Termination Condition**
   - **Compute size of S**
3. **Base Value Constructor G**
   - **Test for equality**

The program description has thus been expressed in terms of a mapping representation. The agenda in the figure above reflects the pending tasks for transformation. There are some tasks at this point which can be carried out immediately by direct transformation of the constructs into their definition. One of them is computing the median of an enumeration of integers:

"The median of an enumeration of integers is the floor of the average of the largest and smallest elements in the enumeration"

This rule obtains the median according to its standard definition. It is expressed in the system as:

'median(X)' & X:enumeration
⇒
'floor(mean(X.lower-bound; X.upperbound))'

The next locus for transformation is determined by the system to be the representation for the mapping, as all other subgoals depend on this one for their completion.

§10 Enumerable Mappings can Transform into Arrays

The algorithm has taken a clear form now. There are several ways of representing a mapping: it can be represented as being a set of pairs, a large tuple, a restricted relation, etc. The system must now aid in choosing an adequate representation for implementing the algorithm efficiently. The first step is to figure out in what ways is the mapping being used. The system uses its internal description mechanisms for this. For any object represented the system stores explicitly all relations in which that particular object is involved. The system provides full cross-referencing as part of the description base and in a manner completely transparent to the rest of the system and the system user.

In this case, the main operations done on the mapping are *restriction*, *computing the size of its domain* and *obtaining its range*. Of these, computing the range is direct for a stored mapping since the range is a structural attribute for this data type. Notice further that the mapping is a mapping from an enumeration into some range set, and that the restrictions of the mapping are to proper sub-enumerations of its domain.[3] The following set of rules expresses this programming knowledge.

> *"A suitable representation for a mapping from an enumeration or enumerable set to another set, is an array whose indices range over the enumeration or the ordinals of the elements of the enumerable domain set, whichever is the case"*
>
> *"To apply the preceding rule on a mapping with an enumerable set as domain, the mapping has to be represented by a mapping from elements of the domain into ordinals and an array of ordinals as indicies which maps into the range of the original mapping"*
>
> *"Subranges of this mapping correspond to subarrays of the array representation"*
>
> *"Contiguous subranges correspond to array regions which can be denoted by pairs whose first coordinate is the array and whose second coordinate is a pair of top and bottom indices of the region."*

These rules propagate effects on the description base and transform substantially the appearance of the evolving program. After transformation of the mapping and replacement of the mapping restrictions by subarray representations the algorithm looks like:

```
function search  (x : integer;
                  S : array [1:size(S)] of integer) : boolean ⇔
if size(S)=1 then S(1)=x
  else  let m be S(⌊(1 + size(S))/2⌋) do
        let S1 = ⟨S, ⟨ 1, m ⟩⟩
            S2 = ⟨S, ⟨ m + 1, size(S) ⟩⟩
        do if x ≤ m
               then search(x,S1)
            else search(x,S2);
```

The system now detects an inconsistency in the recursive call. While the formal argument list uses an array, the recursive call is passing a subarray. To enforce consistency it observes that an array is the same as a subarray of itself which shares

[3] I.e. to contiguous enumerations completely contained in the domain of the mapping.

with it upper and lower bounds. The top level call or interface to the system can then be replaced by an auxiliary function which transforms the call to the second functions format. This new function is introduced and provides the desired interface with the function we now have which is the workhorse for the algorithm.

§11 Dataflow Analysis is Useful for Simplification

An important task in an environment is a facility for simplifying specifications and programs as much as possible as a device for managing complexity and reducing the search space. Data flow plays an important role in this. Most of the direct simplications that can be performed on a program description are data flow invariant. That is, a piece of description is replaced by a functionally equivalent and hopefully simpler description. Data flow facilities in a programming environment enable testing for invariance and inconsistency. For example in the binary search case which we are now considering, several variables which are being bound and only used applicatively can be eliminated by substituting their bindings in the place they appear.

"Bound variables in a region of a specification which are bound and used only applicatively in the region may be eliminated from a program description by substitution of their bound variables in the places where they are used"

The result of all these manipulations takes the system into the following state:

```
function search  (x : integer;
                          S : array [1:size(S)] of integer) : boolean ⇔
search1(x, ⟨S, ⟨ 1, size(S)⟩⟩)

function search1  (x : integer; R : subarray) : boolean ⇔
if size(R)=1 then R.1[1]=x
   else  let m be S($\left\lfloor \frac{1 + size(R)}{2} \right\rfloor$) do
              do if $x \leq m$
                    then search(x, ⟨R.1, ⟨ 1, m ⟩⟩)
                 else search(x, ⟨R.1, ⟨ m + 1, size(R) ⟩⟩;
```

where the *subarray* type is a type definition that has to be introduced dependent on the array type definition in the main function. This is left pending to the refinement of the array type which is still defined in terms of itself. The system has to eliminate the recursion by forcing the size of the argument array to be a constant or a parameter. In order to linearize the type definition it asks the user for the maximum value of the size to introduce a new type. Two more transformations are necessary to produce a complete version of the program implementing the search algorithm. These are:

avoiding passing down the array component of the tuple parameter (since it is not side-effected and can be accessed globally); and making the pair of indices into the array be separate arguments.

Section X

Results in Knowledge-Based Program Synthesis

C. Green, R. P. Gabriel, E. Kant, B. Kedzierski,
B. McCune, J. Phillips, S. Tappel, and S. Westfold

Abstract: This paper presents the current status of the PSI Program Synthesis System. PSI is a system that synthesizes programs from several types of abstract specifications using a knowledge base of rules, producing efficient target-language code This paper is the most recent, complete overview of the entire PSI program synthesis system. It summarizes progress made on the PSI program synthesis system during the past two years. Because of size constraints of this paper, explanation of the detailed internal operation of the system is omitted. For an overview of prior work see [Green-76] and for more detail see [Barstow-79], [Ginsparg-78], [Steinberg-79], or [Kant-79] Following a brief summary of the PSI program synthesis system, a discussion of PSI's present capabilities is given, along with an example program demonstrating its performance. Publications by the Knowledge Based Program Synthesis Group are listed at the end.

Summary of the PSI Program Synthesis System

The PSI program synthesis system is a computer program that acquires high level descriptions of programs and produces efficient implementations of these programs. PSI's operation may be conveniently factored into two parts: the *acquisition phase*, which acquires the model, and the *synthesis phase* which produces a program from the model. Simple symbolic computation programs are specified through dialogues between the user and the PSI system. The specification techniques available include natural language, input-output pairs, partial traces, and a high level specification language, which is being developed at the present time. These specifications are integrated in the program net and model. It is also possible to specify a problem directly in the model language, when this is convenient. The programs produced are in LISP, but experiments have shown that the system can be extended to produce code in a block structured language such as PASCAL. The acquisition phase is similar to the work done by Simon in UNDERSTAND, but PSI emphasizes the synthesis of efficient programs.

This paper is substantially the same as one that appeared in the Proceedings of the 5th IJCAI Conference, Volume 1, Stanford, California, August, 1979.

The overall design is a group effort, with one individual having responsibility for each module as follows: parser/interpreter, Beverly I. Kedzierski; trace and example inference expert, Jorge Phillips; explainer, Richard Gabriel; domain expert, Jorge Phillips; program model builder, Brian McCune; and efficiency expert, Elaine Kant Additional personnel have been working on various projects within these experts. Steve Tappel wrote the rule expander for the program model builder; Stephen Westfold enhanced the examples component of the inference expert. A description of the interactions of the phases and modules of PSI can be found in [Green-77].

§1 Recent Progress on and Present Capabilities of PSI

The entire PSI system can now be used by a knowledgeable user. A number of tests of the entire system have been made. PSI successfully produced LISP code which implements the specifications given in English dialogues for four variants of CLASS, a simple pattern classification program

CLASS is a simple program that includes many programming constructs necessary for more complex programs. The program specified first inputs a sample set of items (e.g., a list of job qualifications). Then the program repeatedly inputs a trial set (e.g., the qualifications of an applicant), tests whether the sample set (required qualifications) is a subset of the trial set (applicant qualifications), and prints "FIT" or "DIDN'T FIT" accordingly. The program is exited if the word "QUIT" is typed instead of a trial set.

§2 Acquisition Phase

The following dialogue was held with PSI to specify CLASS.

```
  PSI: What is the name of the program you want to write?
USER:  CLASS.
  PSI: Does CLASS have a special program type?
USER:  Classification.
  PSI: Please describe CLASS.
USER:  CLASS first inputs a concept.  Then it repeatedly accepts
       an input scene from the user, classifies it based on
       whether or not it fits the concept,
       and outputs this classification to the user.
  PSI: What is the structure of the concept?
USER:  A concept is a collection of relations.  Each relation is
       just a string.
  PSI: What is the exit test of the loop?
USER:  Stop when the user types "QUIT" instead of a scene.
  PSI: What is the structure of the scene?
USER:  The same as the concept.
  PSI: Please define what it means for a scene to fit a concept.
USER:  A scene fits a concept if all the relations in the concept
       are in the scene.
  PSI: What is the format of the classification output?
USER:  Output "FIT" if the scene fit, "DIDN'T FIT" if it didn't.
  PSI: Are we finished?
USER:  Yes.
```

In addition, ten other dialogues have been understood by the parser/interpreter. About three versions of each of five other programs have been coded (some, but not all, of these corresponding to dialogues processed by the parser/interpreter).

A program net is then produced by the parser/interpreter [Ginsparg-78], based upon its understanding of the dialogue. The following description is a summary of this net, the algorithmic part being printed in an ALGOL-like notation.

```
A2 is either a set whose generic element is a string
   or a string whose value is "quit".
A1 is a set whose generic element is a string.
A4 is the generic element of A1.
A3 is either TRUE or FALSE.
B1 is a variable bound to A2.
B2 is a variable bound to A1.
B3 is a variable bound to A4.

CLASS
   PRINT("Ready for the CONCEPT");
   A1 ← READ();
LOOP1:
   PRINT("Ready for the SCENE");
   A2 ← READ();
   IF EQUAL(A2,"quit") THEN GO,TO EXIT1;
   A3 ← FIT(A2,A1);
   CASES:  IF A3 THEN PRINT("fit")
      ELSE IF NOT(A3) THEN PRINT("didn't fit");
   GO,TO LOOP1;
EXIT1:

FIT(B1,B2)
   FOR,ALL B3 IMPLIES(MEMBER(B3,B2),MEMBER(B3,B1));
```

The parser/interpreter now understands over seventy programming concepts and has a vocabulary of more than 175 words. Its programming concepts include data structures (e.g., sets, records), primitive operations (e.g., input, membership), control structures (e.g., loops, conditionals, procedures), and more complicated algorithmic ideas (e.g., user-program interchanges, set construction, quantification). The parser/interpreter is capable of understanding most dialogues which lie within the scope of its concepts and vocabulary. User syntax is not an issue because the parser efficiently parses a very large grammar. The system can sometimes determine the meaning of unknown words (e.g., what concept they represent) from the context in which they appear. The dialogues which the system has understood include those specifying many variants of CLASS, several variants of NEWS (a news story retrieval program), TF (a learning program that uses CLASS as a subroutine), and graph reachability.

The dialogue moderator [Steinberg-79], is capable of choosing which question posed by the parser/interpreter to ask. It has mechanisms (not yet interfaced to the rest of PSI) to answer the question, "Where are we?", and most of the mechanism exists to handle a request to change topic. The moderator has handled dialogues for NEWS and variants of CLASS.

The questions which are asked of the user are quite readable and coherent. Questions use the same terms as the user did in previous sentences of the dialogue. For example, rather than asking for the definition of "A0018", PSI now asks what it means for "a scene to fit a concept". This question generation system has been used in the dialogues for CLASS, NEWS, TF, and RECIPE (a recipe retrieval program similar to NEWS but easier to understand). It has produced about twenty substantially different sentence types. The question generator is being expanded into a more general explainer which will explain PSI's understanding of the program specification given by the user.

PSI will allow programs to be specified by the use of traces and examples. A version of the trace component of the inference expert was completed which handles simple loop and data structure inference such as that needed for the CLASS and TF dialogues The interface with the parser via the program net has been designed. Implementation is complete except for recognition of when the user is giving a trace rather than continuing the dialogue. The examples component has been greatly improved, and an initial version incorporating our subsetting theories has been implemented. This determines (from an example input-output pair for a certain data object) a suitable program transformation that could have carried the object from its initial to its final state.

An initial version of a domain expert for information retrieval has been implemented. Interfaces with the rest of the system are clearly defined, and a common representational base with the parser/interpreter has been completed. This base, called the program net, has been used by the parser for all the dialogues currently done by the system. The program net has also been used in conjunction with the domain expert for the generation of a variant of NEWS.

The program model builder [McCune-77], uses the program net produced by the parser/ interpreter to construct a complete model of the program. From the internal representation of the resulting program model, the understandable model printer produces readable form. The model is printed in a very high level language, similar to PASCAL, but without the usual programming semantics.

A second version of the program model builder has been implemented. Its rule base has increased to 350 rules. The new rules incorporate knowledge of correspondences (or mappings) and primitive operations for accessing them, of procedures and procedure invocations, and of type coercion. The model builder also resolves type-token ambiguities and transforms expressions to canonical forms. A number of program models which are variations on CLASS have been built as part of the entire PSI system. Separately the model builder has successfully constructed the more complex model for RECIPE.

§3 Synthesis Phase

The program model is refined into target language code by the coder [Barstow-79] and efficiency expert [Kant-79A]. Dividing PSI into two separate phases allows programs to be optimized by taking different runtime environments into account. The program can be specified once and a program model built. Then by giving different size estimates, probabilities, or cost functions, different target language programs can be produced.

The programs will of course have the same input-output behavior, but the code will be optimized differently based on the data structure sizes or other such parameters.

Recall that CLASS reads a sample set of items, then repeatedly inputs a trial set and tests whether the sample set is a subset of the trial set. Since the universe of the sets is not known, a subset test using a bit map, which would be very fast, is not possible. So the subset test is implemented as an enumeration through the elements of the sample set, testing each element for membership in the trial set. When the trial set is small, a simple list (the same as the input format) is a good choice of representation for the sets

When the trial set is large, however, it may prove more efficient to convert its represention to a hash table format so that the membership test is much faster. PSI must check whether such savings outweigh the cost of the representation conversion.

The knowledge base of the coding module has grown to about 450 rules. These rules have been used to code a variety of programs involving graph reachability and prime number finding. The sets and correspondences used in these programs can be represented as lists, arrays, Boolean mappings, or property lists. Several versions of CLASS, RECIPE, NEWS, and TF have been coded. Insertion and selection sorts have also been coded. Rules about reusing the space in arrays have been written and used to synthesize in-place selection and insertion sorts (see the section on "Coding an In-Place Insertion Sort"). Some unnecessary variables in the target code are now eliminated by recomputing previously stored results. This can reduce the number of program variables by a factor of two.

The efficiency expert was used with the coder to write five variants of CLASS, to write RECIPE, to write a part of TF, and to write insertion and selection sorts. In all cases different implementations are selected when different data structure sizes (for example) are assumed. More than one representation for the same data structure can be used in a program. There are now rules that suggest the circumstances under which various representations are plausible or implausible. This greatly reduces the search space from the original space of all legal programs. Space-time cost estimates are used to compare alternative plausible alternatives. Cost estimates are also used to identify the decisions that may have the greatest impact on the global program cost; the decision making resources are allocated accordingly.

§4 Additional Results

A number of simulated dialogues have been gathered, with a member of the PSI Group playing the role of PSI and people not part of the group as users. The question choosing algorithm of the dialogue moderator is currently being tested by comparing its behavior with the data from these dialogues.

Preliminary designs have been completed for an additional program specification technique. It is a formal system with the flavor of a very high level programming language.

The language allows manipulation of abstract algebraic structures such as mappings and sets. The semantic support available through the domain expert will allow the use of domain specific jargon in this language. This language will allow the user to specify quickly and precisely program descriptions that have already been well thought out.

A system has been written which prints concise, understandable versions of program models in a PASCAL-like notation. The internal representation of the model is designed for programming efficiency and is hard for people to understand. Listings in the concise notation are thus extremely valuable for debugging. Any or all of the parts of a model may be printed, and cross-reference tables are available to index the concise listing and the original model. Listings may be generated for online viewing or printed out for use in documents.

The program model interpreter, which executes models interpretively as an alternative to coding them and running the target program, has been brought completely up to date with the changes to the program modelling language. It has correctly interpreted the ten program models available. The interpreter can now handle the general case of an input statement in which the datum to be input may be of any type occurring in a tree of legal types.

A comparison was made of the running times of interpreted program models versus corresponding compiled LISP functions coded by the PSI synthesis phase. The functions coded by PSI ran up to eleven times faster than the interpreted models for very simple programs. We expect that time savings will grow more than linearly with program complexity.

A rule expander for model building rules is complete, making it easier to write new rules for the program model builder. Rule preconditions are written in a concise declarative language; then the rule expander translates the declarative form into the required fetch and test operations, taking into account any ordering constraints which the preconditions may have and avoiding retesting preconditions unnecessarily.

§5 Bibliography

[Barstow-79] Barstow, David, *Knowledge Based Program Construction*, Elsevier North Holland, 1979.

[Ginsparg-78] Ginsparg, Jerrold M., *Natural Language Processing in an Automatic Programming Domain*, PhD thesis, Report STAN-CS-78-671, Computer Science Department, Stanford University, June, 1978.

[Green-76] Green, Cordell, "The Design of the PSI Program Synthesis System", *Proceedings Second International Conference on Software Engineering*, Computer Society, Institute of Electrical and Electronics Engineers, Inc., Long Beach, California, October, 1976, pp. 4-18.

[Green-77] Green, Cordell, "A Summary of the PSI Program Synthesis System", In *Proc. IJCAI-77*, MIT, Cambridge, Mass., August, 1977, pp. 380-381.

[Green & Barstow-77] Green, C. C., and Barstow, D. R., "A Hypothetical Dialogue Exhibiting a Knowledge Base for a Program Understanding System", in Elcock, E. W., and Michie, D., editors, *Machine Intelligence 8: Machine Representations of Knowledge*, Ellis Horwood, Ltd., and John Wiley and Sons, Inc., New York, New York, 1977, pages 335-359.

[Green & Barstow-78] Green, Cordell, and Barstow, David, "On Program Synthesis Knowledge", *Artif. Intell.*, 10:3, (1978) 241-279.

[Green & McCune-78] Green, Cordell, and McCune, Brian P., "Application of Knowledge Based Programming to Signal Understanding Systems", *Distributed Sensor Nets: Proceedings of a Workshop*, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, December, 1978, pp.115-118.

[Kant-79A] Kant, Elaine, *Efficiency Considerations in Program Synthesis: A Knowledge Based Approach*, PhD Thesis, Computer Science Department, Stanford University, (in progress).

[Kant-79B] Kant, Elaine, "A Knowledge-Based Approach to Using Efficiency Estimation in Program Synthesis", *Proc. IJCAI-79*, Tokyo, Japan, August 1979 (to appear).

[McCune-77] McCune, Brian P., "The PSI Program Model Builder: Synthesis of Very High Level Programs", **Proceedings of the Symposium on Artificial Intelligence and Programming Languages**, *SIGPLAN Notices*, 12:8, *SIGART Newsletter* No. 64, August 1977, pp. 130-139.

[Phillips-77] Phillips, Jorge V., "Program Inference from Traces Using Multiple Knowledge Sources", *Proc. IJCAI-77*, MIT, Cambridge, Mass., August 1977, page 812.

[Shaw et al.-75] Shaw, David E., Swartout, William R., and Green, C. Cordell, "Inferring LISP Programs from Examples", *Proc. IJCAI-75, Artificial Intelligence Laboratory, MIT*, Cambridge, Massachusetts, September, 1975, pp. 260-267.

[Steinberg-80] Steinberg, Louis, *A Dialogue Moderator for Program Specification Dialogues in the PSI System*, PhD thesis, Computer Science Department, Stanford University, August, 1980.

END
DATE
FILMED
11-81
DTIC