Equational and Rule-Based Programming: Visualization, Reliability, and Knowledge Base Generation

MS-CIS-91-60

Jee-In Kim

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

September 1991

Equational and Rule-Based Programming: Visualization, Reliability, and Knowledge Base Generation

TECHNICAL REPORT

August 1991

Jee-In Kim

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania

Prepared Under Contract AFOSR-90-0335A
from the
Air Force Office of Scientific Research
Bolling Air-Force Base, DC 20332-6448

Equational and Rule-Based Programming: Visualization, Reliability, and Knowledge Base Generation

DISSERTATION PROPOSAL

August 1991

Jee-In Kim

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania

Advisor:
Noah S. Prywes

Committee:
Insup Lee
Tim Finin

Contents

Chapter 1

INTRODUCTION

1.1 Research Problem

This document concerns developing a software development environment for effective use of equational and rule-based programming paradigms. These programming paradigms have significant advantages for developing programs which have requirements for (1) reliability, (2) parallelism, and (3) accumulation of expertise in knowledge bases. The research is motivated by the following questions:

(1) Why equational and rule-based programming paradigms are not used more widely even in areas where they have significant advantages?

(2) What is the software development environment that will make these paradigms attractive to various communities of users?

We hypothesize that a requisite environment for equational and rule-based programming should support (1) visualization of inherent algorithms to facilitate users' composing and understanding of programs, (2) consistency checking of definitions and references of variables, (3) accumulation of expertise in knowledge bases, (4) software testing, and (5) program verification.

To state the research problem, this section presents advantages and disadvantages of three programming paradigms: procedural paradigm which is currently the most popular, functional/equational paradigm which is advantageous for reliability and parallelism, and rule-based paradigm which is advantageous for accumulating expertise. This is followed a brief summary of the proposed software development environment and by a plan for its realization.

A *procedural* (or *imperative*) paradigm employs procedural languages such as COBOL, FORTRAN, and C. Procedural languages are tightly related to the popular Von Neumann machine architecture for sequential computers. A procedural language is based

on *states* that are represented by the contents of the memory, *control* statements for order of the execution, and *assignment* statements which modify the states.

The advantages of this class of languages can be summerized as follows:

(1) **Effective use of sequential computers**: The Von Neumann machine is effectively utilized by executing procedural language programs. Compilers for procedural languages produce efficient code.

(2) **Substantial programming environments**: There are powerful tools for libraries, testing, etc. for these languages.

For these reasons, most of current software has been written in procedural languages.

The disadvantages may be summerized as follows:

(1) **Side-effects**: The procedural style of programming results in side-effects that makes programs difficult to understand [Bac78, PP83].

(2) **Software testing**: Testing to assure program reliability usually involves complex path analysis [Wey86, How87, Bei90].

(3) **Parallelism**: Implementation of parallelism in programs requires eliminating the sequential orientations, much analysis, and intuition [Hud89, Szy91].

(4) **States**: The state-based computation is performed in a "word-at-a-time" style which forces a programmer to "think like a computer" [Bac78].

(5) **Program verification**: The state-based computation model makes it complicated to reason about programs [Bac78, Hud89, Szy91]; there exist formal verification methods for procedural languages [BM81, MP81, OL82, Lam83, HM85, Hoa85, CES86, Lin88, Mil89, CPS90] but they are too complicated to be widely used by programmers, mainly because dealing with program states is complicated.

As a result, it is claimed to be difficult to guarantee a high degree of reliability for procedural language programs.

Functional/equational languages employ algebraic definitions of variables. The value of a variable is uniquely defined and is not modifiable. The computational model of equational languages is based on regular and boolean algebras [Hud91, Szy91]. There are no implicit states, no side-effects, and no predefined sequence of computation. Equational languages such as EPL[Szy91], Haskell [Hud91], and MODEL [PP83, SP88, Hud89] are a special case of functional languages. They are characterized by the followings:

(1) **Equational look**: A statement looks like an equation.

(2) **Referential transparency** [Hud89, GJ90]: A variable has a unique value defined by an equation in equational languages. Once the value is defined, it is not modifiable.

(3) **Equational reasoning**: It is possible because of referential transparency to employ equational reasoning for program verification [Hud89, Szy91, Hud91]. Equational languages can be based on "high school" algebras such as equivalence laws, substitution, transitivity, and induction [Gri81]. Thus the equational reasoning method for program verification can be easily understood and performed by ordinary programmers. The equational reasoning is not only for reasoning about programs but also for writing and debugging programs without considering implementation details [Hud89].

(4) **Iterative definition**: Data elements of arrays can be defined iteratively in equational languages. Recursive definitions can be used (as in pure LISP), but they may be more difficult to analyze and reason about.

(5) **Program visualization**: The execution of an equational language program can be visualized with the aid of Petri-net like graphs. The nodes denote equations and data elements and the edges denote data flows among the nodes. Functional languages, where program execution is recursive, form a tree structure. They may be visualized by a tree where its root node denotes the main function of a program, its children nodes denote the subfunctions called by the main function, and its edges denote function calls.

The advantages of use of an equational language may be summerized as follows:

(1) **Use of well known algebraic equations**: An equational language is based on algebra which is widely known. While the paradigm of programming in this manner is new, it can be easily taught to attain wide usage.

(2) **Parallelism**: An equational language paradigm is not biased by the need for sequential execution. Therefore it is easier to create compilers that map equations into parallel processors [Szy91, CCL91].

(3) **Reliability**: As will be argued later, a thorough testing methodology for improving reliability of programs is easier to implement. Also, verifying correctness using equational reasoning is easier to employ and teach.

The disadvantages of equational languages may be summerized as follows:

(1) **New paradigm**: The great numbers of programmers are familiar with procedural languages and may decline to learn a new paradigm and/or be unwilling to change their mode of programming. This is even more so as in some cases procedural language programs may be shorter.

(2) **Complexity of equational language compilers**: Smarter compilers are necessary which optimize use of memory and allocate computations of equations to parallel processors. Such compilers are being developed [Lu81, SP88, Bru89, CCL91, Szy91]

Rule-based languages employ a set of rules to form a program which is executed by an expert system. Each rule in a program has *preconditions* that must be satisfied before executing the rule and *actions* that specify the consequences of the execution. As soon as the preconditions are satisfied, the corresponding rules are fired to perform the specified actions. Rules represent expertise in certain areas and can be accumulated in knowledge bases [GR89]. A rule-based expert system is the underlying computing system that consists of *knowledge base*, where the expertise is accumulated, and *inference engine*, which actually executes the rules of the system to exercise the expertise.

The advantages of the rule-based paradigm can be summerized as follows:

(1) **Incremental augmentation**: Knowledge bases can be incrementally augmented through adding a consistent and reliable set of rules.

(2) **Interactive explanation**: The rule-based paradigm provides man-machine interaction for "explaining" the results of computation. For example, an expert system can explain how a decision was reached by revealing the inferences that were employed. As will be shown, this capability can be used to test programs to attain higher reliability.

(3) **Forward and backward chaining**: These capabilities of expert systems enable users to compute "outputs" from "inputs" as well as in some cases "inputs" from given "outputs".

The disadvantages may be summerized as follows:

(1) **Quality of expertise**: The success of rule-based programming depends on the quality of expertise in knowledge bases. In general, it is difficult to collect expertise. There are difficult procedures for extracting expertise rules from human experts and organizing the rules into knowledge bases.

(2) **Reliability of expertise**: It is difficult to define the domain where the rules in the knowledge bases can be fully relied on for optimal decisions.

Equational and rule-based languages have not been popular in solving "real world" problems in spite of their distinctive merits over procedural languages. We believe that a lack of programming tools and environments, especially in the areas of visualization, knowledge base generation, testing, and verification, is one of the main reasons why the software industry has not been more active in taking advantage of these languages. Our research problem is to define a programming environment for an equational language

and a rule-based language where programmers (1) understand and compose programs with ease, (2) produce reliable programs, and (3) extract expertise embodied in existing and new programs for augmenting knowledge bases.

1.2 Contributions

A software development environment for equational and rule-based paradigms is proposed in this document. The aim is to make it easier for a programmer to understand and use these languages, compose more reliable programs, and automatically enrich rule-based knowledge bases with the expertise in existing programs. We believe that the merits of programming in equational and rule-based languages will be recognized if programmers are equipped with such an environment.

Figure 1.1 provides an overview of the proposed environment. It provides interactive tools for visualization, consistency checking, knowledge base generation, software testing, and program verification. The graphical user interface controls interactions between programmers and all the tools. The repository of the environment contains programs, requirements, graphs, equations, proofs etc.

Chapter 2 provides introductory theoretical foundations of the proposed environment: syntax, semantics, an execution model, and a calculus for manipulating an equational language.

Subsequent chapters discuss the tools of the proposed environment. The innovative aspects of the tools are summerized as follows:

(1) **Visualization**:

Visualization of programming activities has the following advantages over the conventional (textual) programming style [Mye88, Cha90]: (1) The capability of human beings for visual information processing can be utilized. (2) Graphical representation of programming is helpful for human beings in composing and understanding programs, especially non-programmers or novice programmers. (3) Understanding of complex problems, such as concurrent processes and real-time systems, can be enhanced through visualization.

There are two notions of visualization: *program visualization* that reveals some aspects of programs in graphical forms such as graphs, diagrams, and charts and *visual programming* that allows programmers to compose programs using graphical objects such as icons [Mye88]. The proposed environment provides tools for both program visualization and visual programming.

The visualization tool produces a graphical user interface for all the other tools: for consistency checking, knowledge base generation, software testing, and program verification. The graphical user interface provides programmers convenient and easy accesses to the tools which may be too complicated to be used otherwise.

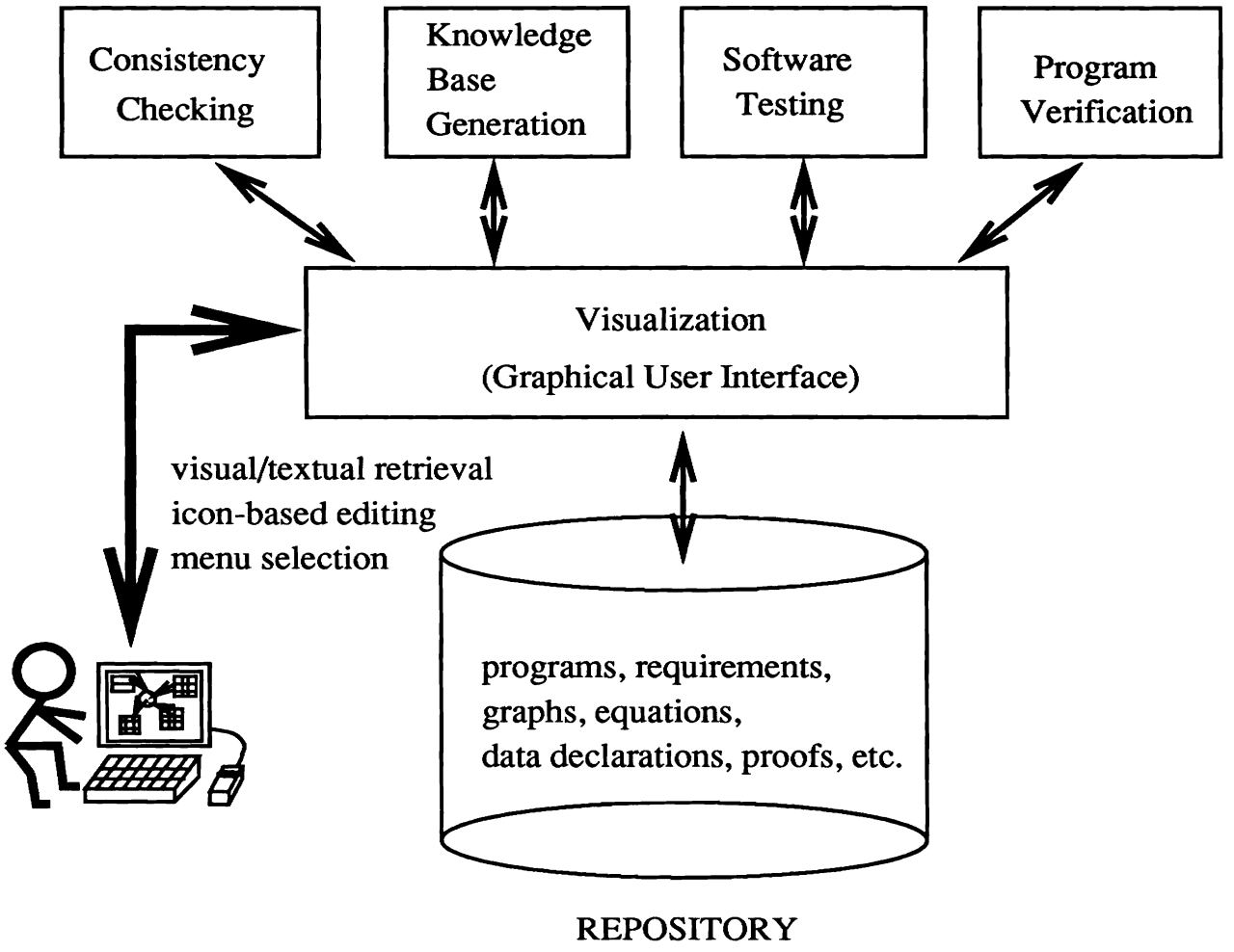


Figure 1.1: A software development environment for equational and rule-based paradigms

The basic graphical object of the visualization of equational and rule-based programs is a *Petri-net like data flow graph*; its nodes denote equations and data elements; its edges denote data flows among the nodes. These graphs are referred to *array graphs* [Lu81, PP83]. These graphs facilitate the visualization much better than *flow charts* of procedural language programs. The visualized data flow graphs can be piecewise examined due to referential transparency, while the whole flow chart of a procedural language program must be examined due to possible side-effects of changes of program states. It has been argued that flow charts are a poor abstraction of software structure and useless as a design tool [Bro87].

Programmers will construct the Petri-net like graphs using an icon-based graphical editor. Textual definitions for the detailed descriptions of the computation will be associated with the graphs. While the graphs are displayed in graph windows on a screen, the textual definitions are given in text windows. By selecting proper operations from pull-down menus using a mouse, programmers can perform a variety of tasks, interactively exercising both visual and textual information.

There are also many *graphical operations* such as for managing windows, exploding/imploding nodes, zooming, taking a snap-shot, etc. They are selected from pull-down menus.

The graphical user interface and operations for program visualization are described in Chapter 3. An icon-based graphical editor is proposed for visual programming.

(2) **Consistency checking**:

The checking mechanism of the environment aims to facilitate users composing an equational or a rule-based language program in accordance with the semantics of the languages. It utilizes techniques developed for equational language compilers [Lu81, Bru89, Set89]. If errors are detected, the mechanism generates error messages. The mechanism also generates warning messages that spell out implicit assumptions made because of omissions. The checking mechanism is combined with the icon-based graphical editor so that programmers interactively exercise composing and debugging their programs. It is one of the unique features of the proposed environment. Chapter 4 describes the checking mechanisms.

The checking mechanism is based on the semantics of these languages: One requirement is for existence of a *causal chain* that computes a solution set for a given input values. A cyclic definition of variables may cause an infinite computation. The checking mechanism detects such a cycle in Petri-net like graphs and tests if it represents a cyclic definition.

It is required in equational languages that every variable must exist and be defined. Such an existence requirement is also checked. It is examined by checking the definitions of variables and their references.

As the Petri-net like graph is examined, a table of variables and their definitions is constructed. The table is used by the checking mechanism [Lu81]. It aims to detect ambiguous definitions, incomplete definitions, and data type mismatches of the variables and the equations.

There may be omissions or discrepancies in declarations and references of variables about dimensionalities, or ranges of arrays referenced in subscript expressions. They are checked by propagating attributes such as dimensions and ranges[1] via edges of a Petri-net like graph.

The condition of terminating the execution of iterations can be checked too. Even though there is no algorithm to decide program termination in general, the checking mechanism can statically check the termination condition and generate the conditions for iterations in warning messages Programmers utilize such an information displayed visually in the graphs in order for composing reliable programs.

(3) **Knowledge base generation and software testing**:

There is valuable expertise in existing programs that can be automatically translated to rules in order to enrich knowledge bases of rule-based expert systems. The proposed environment will allow users to extract such expertise from existing programs and accumulate it as rules in knowledge bases. The expertise, such as algorithms and methods in programs, will be automatically translated to rules of expert systems. The notion is to have two translation steps: (1) use of an existing method [Lu81, GP89] to translate procedural language programs into equational language programs and (2) translation of the equational language programs into rule-based language programs. Testing will be performed on the rule-based language programs using an expert system in order to increase the program reliability. Verification will be also exercised to assure a higher degree of the reliability. Through the translation steps, the expertise in programs can be transferred to knowledge bases, tested, and verified. We believe that the translation, testing, and verification will reduce the labor of collecting expertise for knowledge bases.

Software testing aims to discover faults in a program by executing it with test input data [DMMP87, Ham88, Bei90]. This enhances the reliability of programs. The procedures required for software testing in the procedural paradigm are complicated and tedious. Every paths of changing values of each variable must be tested. Data flow analysis has been claimed to be more effective and powerful [RW85, How86, How87, Wey90]. The data flow analysis in procedural paradigm requires to test every path between definitions of each variable and its references [RW85]. Since the procedural paradigm allows modification of variable values, we must test multiple paths by which the value of each variable is changing. However, in using equational or rule-base languages, it is sufficient to test only a single path for each variable, because each variable is uniquely defined. It

[1] size of a dimension of an array variable.

greatly reduces the complexity of software testing procedures, especially the data flow analysis. Human testers interactively exercise the testing using an expert system in conjunction with the visualization where programs are executed. Test input and the test results are interactively entered and processed. The expert system not only computes the results from a given set of test input data but also gives detailed explanations about the testing. Moreover it computes "inputs" from given "outputs" using backward chaining. The testing is exercised via the graphical user interface which (1) accepts test inputs via the displayed Petri-net like graphs, (2) shows the status of the testing via the graphs, and (3) displays the results of the testing such as test output data, test coverage, etc. on the graphs. The visual information makes it easier to exercise the testing.

Chapter 5 discusses translation of equations into expert system rules. The translation technique is applied to both the knowledge base generation and the software testing. The software testing methodology via the graphical user interface is also described.

(4) **Program verification:**

A higher degree of program reliability can be obtained by employing *program verification.* It is a process of proving logical assertions about computational properties of programs [BM81, Dij81, OL82, Lam83, CES86, Kro87, BGM90, CPS90]. The assertions usually concern whether or not programs properly perform desired functions specified in their functional requirements. This has todate required a highly trained expert in both mathematics and software engineering.

In equational languages, equational reasoning offers simple and more intuitive ways for program verification utilizing only substitution, transitivity, equivalence laws. We propose an equational reasoning system and a graphical user interface for interactive program verification. It borrows the concepts from *interactive heterogeneous reasoning* [BE90a, BE90b, Shi91], which consists of equational reasoning based on visual and textual information. Visual information about programs such as Petri-net like graphs is combined with textual information such as equations and data declarations during the verification. A human tester dictates the procedures of the verification. He guides the verification system by requesting it to make substitutions and simplify expressions via the graphical user interface. A symbolic manipulator mechanically applies the equivalence laws and the rules of equational reasoning to simplify expressions. The program verification procedure is illustrated in Chapter 6. It is exercised through interaction between a human tester, the symbolic manipulator, and the graphical user interface.

The tools of the visualization, the consistency checking, the knowledge base generation, the software testing, and the program verification, are integrated in the proposed environment. Programmers can employ these facilities to understand and compose reliable programs. Programmers can utilize both visual and textual information in programming, testing, and verification. The environment facilitates automatic augmentation of knowledge bases with expertise extracted from existing programs. We

believe that the environment will greatly reduce the costs and increase the reliability of software development and maintenance.

1.3 Research Plan

The plan to implement the environment includes the following tools:

(1) **An equational language, MODEL:**

MODEL [Lu81, MOD89] is an equational language based on regular and boolean algebras; it has no side-effects, no states, and no control statements. It has an equational look and referential transparency. Its data elements can be iteratively defined. Equational reasoning can be applied for program verification. A Petri-net like data flow graphs can be visualized. It has high level data structures such as arrays and supports structured variables, records, and files. MODEL has been successfully used in various applications of science, engineering and business [PP83]. The research on MODEL includes: language translation [Lu81, MOD89], consistency checking [Lu81, SLPP84], code optimization [SP88], reverse engineering [PLGS88], concurrent programming [PGLS90], etc. Therefore we can take advantage of the MODEL compiler and the theories of MODEL in implementing the proposed environment.

(2) **An expert system, CLIPS:**

CLIPS (C Language Integrated Production System) [GR89, Met91] is a rule-based expert system developed by NASA. It is written in C to support high portability, low cost, and ease of integration with external systems [Met91]. CLIPS can be run under various operating systems such as Unix, VMS, MS-DOS, Macintosh, etc. We will use CLIPS as the rule-based programming language of the environment, mainly because of its availability. A CLIPS rule consists of pre-conditions and actions. Its execution is based on pattern-matching, the Rete algorithm [For82, GR89, Met91]. CLIPS also supports object-oriented programming. A number of applications with graphical user interfaces are built on CLIPS. In the proposed environment CLIPS will be used as knowledge based system for software testing where expertise extracted from programs are accumulated.

(3) **A meta-environment, DECdesign:**

DECdesign [Dec90] is a meta-environment that helps users develop their own graphical environment. It allows users to customize the environment by encoding their own methodologies using MDF (Methodology Definition File) and MIL (Methodology Implementation Language). It also provides tools of creating and managing graphical objects on X Window System. The graphical user interface, the icon-based graphical editor, and the program visualizer will be implemented using MIL. DECdesign provides tools of implementing a repository, such

as managing user accounts, creating libraries, moving data between libraries, etc. [Dec90]. The repository of the environment will be implemented using MIL too.

The implementation of the complete proposed environment will require much effort. At this stage of the research, we plan to implement only a basic part of the environment. This will consist of only the three of the five components in Figure 1.1: visualization, software testing, and knowledge base generation. It is planned that the visualization part is implemented using DECdesign. Since MIL is a very high level language, it would take 6 months to implement the program visualizer and the icon-based graphical editor. The implementation of the CLIPS rule translator is estimated to take 3 months. The total integration of the tools requires a significant amount of time: The MODEL compiler (already implemented), the CLIPS rule translator (to be implemented), and the CLIPS expert system (to be purchased) will be combined with the graphical user interface. We estimate that the integration of the prototype will take 3 months.

Chapter 2

THEORETICAL FOUNDATIONS

2.1 Introduction

MODEL is a high level mathematical language. It can be used for composing *equations* and *declarations* that specify an algorithm and writing its *requirement assertions.*[1] Programmers can compose programs using equations and data declarations to implement the algorithm without considering implementation details. The MODEL specification of the algorithm can be understood by programmers with ease because it is based on regular and boolean algebras that can be learned from high school. Formal verification of the correctness of a MODEL specification is easier than that of a program, because it utilizes only algebraic manipulation of equations and requirement assertions. Equational reasoning cab consist of algebraic laws of equivalence, rewriting rules, such as substitution and transitivity, and an induction rule. This can be used for the MODEL verification. The theoretical foundations of MODEL and its calculus are defined in this chapter, as a basis for composing, understanding, testing and verifying a MODEL specification. Their use in proving correctness of programs is illustrated with a simple example.

Figure 2.1 illustrates the overall approach to program verification based on equational reasoning. The input to the verification system consists of a MODEL specification, and proof goals. The verification system is based on equational reasoning that utilizes algebraic laws, rewriting rules, and an induction rule. As outputs of the system, formal proofs about the proof goals are constructed.

A *MODEL specification* expresses algorithms to solve a given problem in terms of equations. Each *equation* in the specification is considered as an *axiom* in the MODEL calculus. Thus the following requirements must be satisfied:

[1] In practice, requirements of software are usually written in natural language. We assume that a human tester must be able to translate requirements written in natural language or any other forms into respective formal representation in MODEL. From now on, requirement assertions are assumed to be written in MODEL.

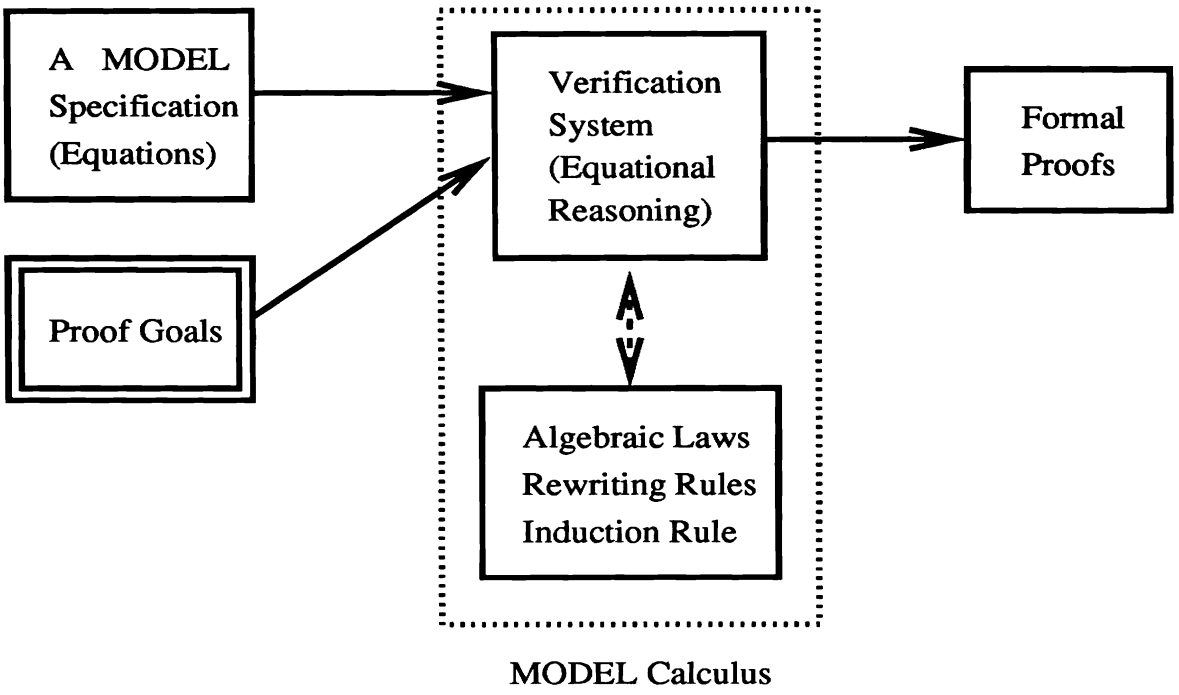


Figure 2.1: An overview of a process of verifying the correctness of a MODEL specification.

(1) A user must assure that all the inputs of each equation are available.

(2) The equation must compute the unique value of its left-hade side (LHS) variable.

(3) There must exist at least one *causal chain* [GR89] from input to output via the equations.

(4) The execution of a specification must terminate with a solution.

The verification process aims to prove the correctness of the specification as presented in the *proof goals*. The proof goals may consist, in the simplest case, of constraints on inputs and their expected outputs. They may consist of assertions (expressed as MODEL equations) about computational properties of the specification. Proof goals are decomposed into a collection of subgoals that can be proven one at a time.

The MODEL calculus has the same syntax of the MODEL language. It uses algebraic equivalence laws and rewriting rules which evaluate and manipulate expressions during the process of verification. The calculus also employs an induction rule.

This chapter is organized as follows: Section 2.2 describes the syntax and the semantics of the equational language. The basic notions of the MODEL calculus, the algebraic laws, the rewriting rules and the induction rule are presented in Section 2.3. An example is given in Section 2.4 of verifying the correctness of a MODEL specification under this calculus.

2.2 MODEL

2.2.1 Functional Units

A *functional unit* in MODEL consists of a *header*, *declaration* and *equations*. A header is an interface of a functional unit and specifies its type (*module, function* or *procedure*), name and a list of inputs and outputs. A multi-unit specification consists of a main functional unit, called module, and a set of subsidiary functional units, either functions or procedures [PLGS88]. A function accepts structures of input parameters and returns a single output structure. A procedure has input, output and update (treated as "*new*" and "*old*") parameters. As will be shown, definitions of subsidiary functions or procedures are in fact definitions of operations. An individual functional unit does not have recursive definition in itself, although it can use itself as an operation thus creating recursion. In the following, we focus on an individual module, function or procedure which are called "programs-in-the-small". For "programs-in-the-large", see [LP90].

2.2.2 Data Declaration

Data structures and their types are declared in a declaration part of a functional unit. There are input, output and interim variables. Input and output variables are declared in its header. A structure of each input and output variable, i.e. an entire hierarchy of the structured variable, must be specified in a declaration part (it may be typed). Interim variables are used within a functional unit and cannot be accessed from the outside. Their declaration is optional. If there is no explicit declaration for an interim variable in a declaration part, a translator from MODEL to a procedural language inserts its declaration automatically. A primitive type of a variable is one of the followings: boolean, integer, real, or literal. The primitive type is defined either explicitly or implicitly in the data declaration. We do not discuss the data declaration further as the focus of the following is on the equations.

2.2.3 Equations

The syntax of an equation is defined as follows [MOD89]:

$$Equation ::= SimpleEquation | ConditionalEquation^2$$

$$SimpleEquation ::= VarName(SubExpr_1, ..., SubExpr_n) = AnyExpression$$

$$AnyExpression ::= ArthExpression | StrExpression | BoolExpression$$

[2] We have three meta-symbols; "::=" defines a term in its left-hand side, "|" denotes **or** and "[*expr*]" means "*expr*" is optional.

$ConditionalEquation ::= VarName(SubExpr_1, ..., SubExpr_n) = CondExpression$

$CondExpression ::=$ **IF** $BoolExpression$ **THEN** $CoAnEx$ [**ELSE** $CoAnEx$]

$CoAnEx ::= CondExpression | AnyExpression$

An equation is either simple (*SimpleEquation*) or conditional (*ConditionalEquation*). They define an LHS variable $VarName(SubExpr_1, ...)$ in terms of expressions. In composing a MODEL specification, only a variable (possibly subscripted) is allowed in the LHS of an equality in an equation.[3] An expression in the right-hand side (RHS) of an equality defines the variable. *BoolExpression* defines a Boolean variable. Integer and real type variables are defined by arithmetic expressions. String expressions return literal values for a string variable.

2.2.4 Array and Scalar Variables

A MODEL variable is either a *scalar* or an *array*. An array variable is indexed by a set of *subscript expressions*. As in mathematics, each variable has a single value in MODEL. Once its value is assigned, it never changes. On the other hand, a *subscript variable* assumes all integer values in the range of the elements of the arrays. Such subscripts are further discussed below.

Every array has a data declaration or an equation that defines its dimensionality, either implicitly or explicitly. For example, an equation $END.x(i) = exp(i)$ may be defined for the range of an array x with a subscript i, where exp is a Boolean expression and a function of i. The variable $END.x(i)$ is called a *control variable* [Lu81, MOD89]. END is prefixed to the array variable x. It is a "shadow" variable of x in the sense that it has the same shape as x. See Figure 2.2. The value (a truth symbol, either $TRUE$ or $FALSE$) of the control variable is defined by the equation $END.x(i) = exp(i)$. The values of $END.x(i)$ are $FALSE$ except for the value of the last element in the most right dimension that has the value $TRUE$. The size of the array variable x can be alternately defined directly by another prefixed control variable, $SIZE.x$. The array variable x is defined only when its subscript i satisfies a predicate $1 \leq i \leq SIZE.x$. If the array is finite, $SIZE.x$ has a finite value. Every element of $END.x$ has the $FALSE$ value while the last element $END.x(SIZE.x)$ is $TRUE$. If x is an infinite array, however, there is no $TRUE$ element in the array $END.x$, i.e., the value of $SIZE.x$ is infinite. It concludes that the following two equations are equivalent:

- **Rule of Control Variables:**

  $(END.x(i) = exp(i)) \equiv (exp(i) =$ **IF** $(i = SIZE.x)$ **THEN** $TRUE$ **ELSE** $FALSE)$

[3]The restriction that only a variable is allowed in the LHS is relaxed by [Ge89]. In his extension, an LHS expression is defined as equal to an RHS expression. We use the extended MODEL language in formulating requirement assertions.

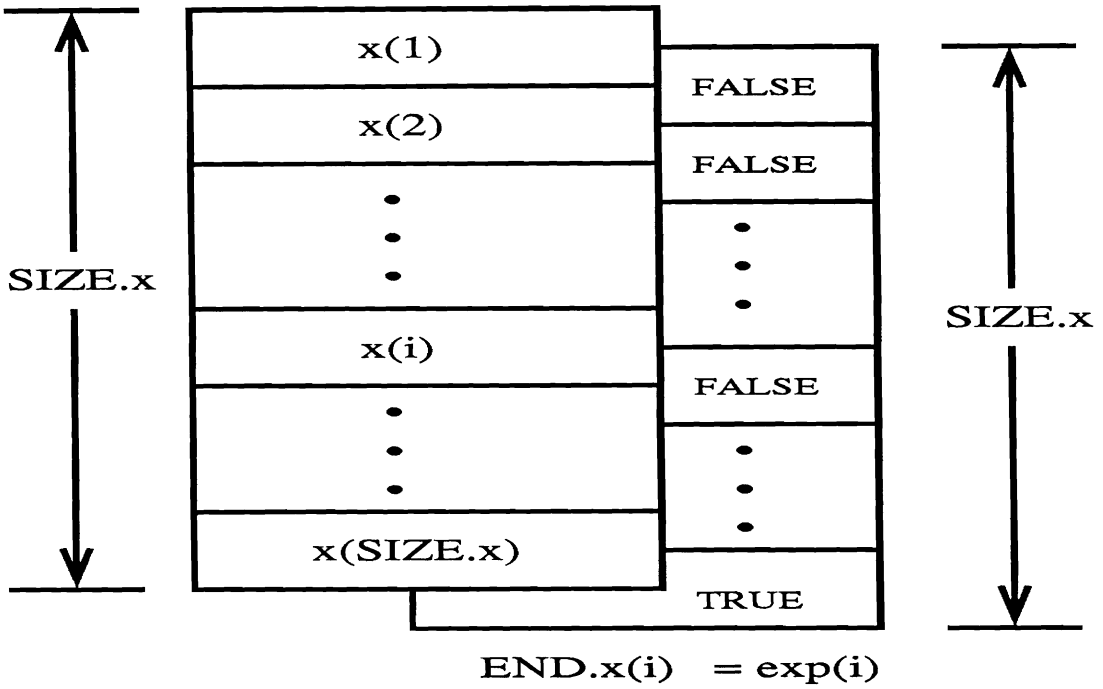


Figure 2.2: An array variable x and its "shadow" array variable $END.x$.

2.2.5 Operations

The expressions of a MODEL specification use a set of operations. The operations are defined by operators, functions and their arguments. The followings are MODEL operators for expressions:

(1) Arithmetic Operators: + (addition), − (subtraction), ∗ (multiplication) and / (division).

(2) Relational Operators: < (less than), ≤ (less than equal), > (greater than), ≥ (greater than equal), = (equal) and ≠ (not equal).

(3) Logical Operator: & (and), | (or), ¬ (not) and **IF-THEN-ELSE.**[4]

(4) String Operator: || (concatenation), string search and string replacement.

Functions are viewed as operations on their input arguments. They are either *built-in* or *user defined.*

2.2.6 Implicit Universal and Existential Quantifiers

The most distinctive difference between a procedural language and an equational language is that a variable has a single value in an equational language such as MODEL.

[4]**ELSE** is optional.

On the other hand, we can change the value of a variable in a procedural language as many times as we want to.

A MODEL equation, $x(i) = x(i-1) + 1$, defines all elements of x indexed by the subscript i. It means that the equation represents a class of all equations such that $\forall i, 1 \leq i \leq SIZE.x$.

The MODEL equation, $x(i) = x(i-1) + 1$, can be interpreted into the following code in a procedural programming language such as FORTRAN:

```
DO I = 1, SIZEX
   X = X + 1
ENDDO
```

In the FORTRAN code, the assignment statement `X = X + 1` cannot be executed if the index variable `I` is out of its range, i.e. either $I < 1$ or $I > SIZEX$. It can be executed only when the index variable is properly defined. Likewise, a MODEL equation is defined only when its LHS variable subscript expression is within the range. If the LHS variable is size 0 or its subscript expression is out of range, the equation is undefined. A MODEL equation that has undefined LHS variable is invalid. Such an equation is called a *null* (or invalid) equation.

There is also the case of a null equation for specific subscript values: a conditional expression without **ELSE** in the RHS of the equation. Suppose we have the following equation: $y =$ **IF** $i = 10$ **THEN** $x(i) + 1$. y is a scalar and its value is defined only for the equation instance of $i = 10$. The equation is invalid for other value of i. However there must be one instance (value of i) where the equation defines y, i.e. y exists. In short, an instance of an equation becomes a null equation if its LHS variable is undefined or its RHS expression cannot define valid operations for its legal LHS variable.

A MODEL specification is regarded as a collection of *valid* equations. A valid equation is defined to be able to **uniquely** determine the value of its LHS variable in terms of its RHS expression. All the subscript expressions of every LHS variable should be defined within their legal range. There exists a unique RHS expression of a valid operation (that determines its value) for each element of an LHS variable. If multiple equations define the same LHS variable, they must be mutually exclusive of each other.

It concludes that existential ($\exists$) and universal ($\forall$) quantifiers implicitly exist for each MODEL equation. Consider an equation

$$x(i_1, ..., i_n) = f(j_1, ..., j_m, var_1, ...var_k)$$

where the LHS variable x is defined by the function f; the LHS variable is indexed by the subscripts $i_1, ..., i_n$ and the function f may have subscripts $j_1, ..., j_m$ and the variables $var_1, ..., var_k$ as its arguments. If the ranges of the subscripts are $1 \leq i_1 \leq$

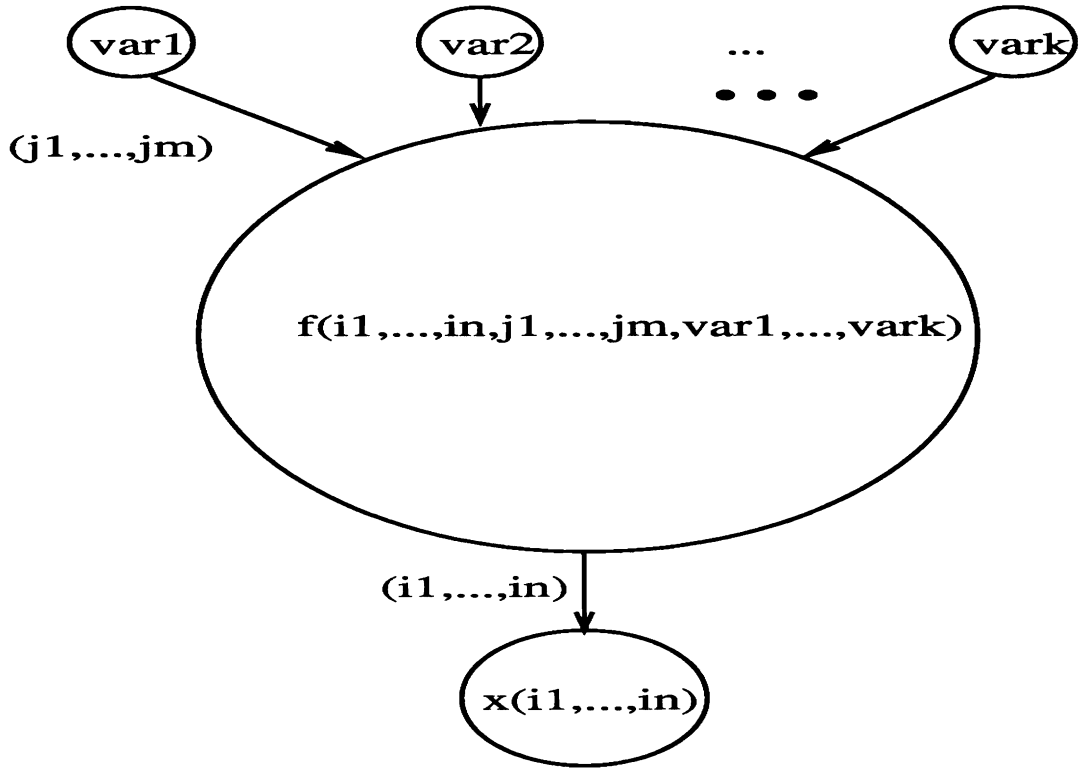


Figure 2.3: The Existence Condition.

$SIZE_{i_1}$, $1 \leq i_2 \leq SIZE_{i_2}$, etc., the following expression is equivalent to the MODEL equation:

Existence Condition of Equation:

$\forall\ i_1, ..., i_n, j_1, ..., j_m,\ ((\exists j_1, ..., j_m, var_1, ..., var_k,$

$1 \leq j_1 \leq SIZE_{j_1}, ..., 1 \leq j_m \leq SIZE_{j_m}, f(j_1, ..., j_m, var_1, ..., var_k)) \Rightarrow$

$(\exists i_1, ..., i_n,\ 1 \leq i_1 \leq SIZE_{i_1}, ..., 1 \leq i_n \leq SIZE_{i_n},$

$(x(i_1, ..., i_n) = f(j_1, ..., j_m, var_1, ..., var_k)))$

The expression is interpreted as follows:

> For all subscripts of the LHS variable, x, and the function, f, there exist the subscripts of the function in the legal range, all input variables of the function are available and the function is computable. Then the equation defines the value of the LHS variable, x.

The existence condition of the equation, in fact, describes one of the requirements that must be satisfied by MODEL equations in a specification: Every variable must

exist and be uniquely defined. The composition of such MODEL specifications is facilitated by the checking mechanism as will be discussed in Chapter 4. Incomplete, ambiguous and/or inconsistent definitions of subscript expressions, variables and equation are assumed to be removed after being detected by the checking mechanism [Lu81, SLPP84]. Any cyclic definition of variables is removed by the checking mechanism.

2.2.7 Execution Model

One way to envisage MODEL specification execution is as a data flow machine [Arv82]. That is, the expression in its RHS is executed as soon as all of the inputs are available. The value of the LHS variable is determined as the RHS expression is evaluated. The notion of execution in MODEL consists of firing each equation when its inputs are available and generating the corresponding output. An equation is fired once for each combination of legal LHS and RHS subscript variables. However, due to the implicit universal and existential quantifiers, only one LHS element is defined for each legal LHS subscript variable.

A MODEL specification may include declaration of input and output data. Due to the termination assumption, for any set of input data, there must exists at least one order of firing that uniquely defines all the LHS variables (scalars or elements of arrays). The specification will be checked to assure that such a sequence exists. Otherwise, the specification is illegal.

Another way to envisage MODEL specification execution is an expert system [GR89]. An equation is implemented as a rule of an expert system. It is expressed as follows:

$$precondition \Rightarrow action$$

Each equation is a rule that has implicit universal and existential quantifiers as its existence condition. It is in the form of the following:

$$existence\_condition \Rightarrow equation$$

Since the equation is fired only if the existence condition is satisfied, it can be interpreted as the *precondition* of the rule. The execution of the equation is regarded as the *action* of the rule. The specification of equations can be viewed as a set of rules. The input data of the specification such as "$x = 3$;" can be interpreted as a set of given *facts* of the expert system such as "`(define-fact (x 3))`". An array variable like "$y(i, j, k) = 47$;" is expressed as a fact (or relation) that maps multiple fields of its subscripts to its value such as "`(define-fact (y i j k 47))`". As the precondition (= the existence condition of the input variables) is satisfied, the values of the equations, the LHS subscripts and variables are determined through *unification*. Thus the action (= the execution of the equation) determines the LHS variable. The execution of a specification is actually the process of inference (or reasoning) on the given facts in the expert system. The order of firing represents *causal chain* (or *reasoning*) among

the rules [GR89]. Since a specification is assumed to have at least one order of firing that leads to a solution, there is at least one causal chain for a given set of facts. The details of the expert system implementation are discussed in Chapter 5.

2.3 A MODEL Calculus

We define algebraic structure, laws and rewriting rules for manipulating MODEL expressions. The basic notions are described in Section 2.3.1. The laws of arithmetic operators, relational operators and logical operators are described in Section 2.3.2. To apply those laws to expressions, we need rewriting rules: Substitution and Transitivity. Section 2.3.3 describes these rules. An induction rule is also needed and specified in Section 2.3.4.

An equality ($=$) is used in MODEL expressions. In defining algebraic laws, an equivalence relation ($\equiv$) is used as a meta-symbol. It is reflexive, symmetric and transitive.

2.3.1 Basic Notions

Well Formed Formula

The calculus shares the same syntax with the MODEL language. That is, legal MODEL expressions and equations are defined as *well-formed formulas* (wffs) of the calculus. The truth symbols, $TRUE$ and $FALSE$, must be wffs. Constants and variables of MODEL are wffs of the calculus. Since the calculus only allows the addition operator ($+$) and the multiplication ($*$) operator as will be discussed, the subtraction operator ($-$) and the division operator ($/$) defined in MODEL are regarded as the inverse operators of $+$ and $*$, respectively. If A and B are wffs, $A + B$ and $A * B$ are wffs. $A - B$ and A/B in MODEL are translated as the wffs of the calculus, $A + (-B)$ and $A * (1/B)$, respectively.

Algebraic Structure

The algebraic structure for the calculus is defined as a *field of fractions* (rational number), $(F, +, *)$ [Gil76]. It has the following components:

(1) F is a set of rational numbers. Its element is defined as a/b, where $a \in I$ (integer) and $b \in I - \{0\}$. An equivalence relation $\sim$ forms equivalence classes such that $a/b \sim c/d \; iff \; a * d = b * c$ in I.

(2) addition ($+$) and multiplication ($*$) are defined as follows:

$$(a/b) + (c/d) = (a * d + b * c)/(b * d)$$
$$(a/b) * (c/d) = (a * c)/(b * d)$$

(3) *identities*: $0/1$ for $+$ and $1/1$ for $*$

(4) *inverses*: an inverse of a/b for $+$ is $-a/b$ and an inverse of non-zero a/b for $*$ is b/a

Semantics of Relational Operators

Relational operators are used in comparing values of expressions. The semantics of the operators are defined as follows:[5]

1. $E_1 = E_1$ is $TRUE$ if the values of E_1 and E_2 are equal and is $FALSE$ otherwise.
2. $E_1 > E_2$ is $TRUE$ if the value of E_1 is greater than that of E_2 and is $FALSE$ otherwise.
3. $E_1 < E_2$ is $TRUE$ if the value of E_1 is less than that of E_2 and is $FALSE$ otherwise.
4. $E_1 \leq E_2 \equiv ((E_1 < E_2)|(E_1 = E_2))$
5. $E_1 \geq E_2 \equiv ((E_1 > E_2)|(E_1 = E_2))$
6. $\neg(E_1 \leq E_2) \equiv (E_1 > E_2)$
7. $\neg(E_1 \geq E_2) \equiv (E_1 < E_2)$
8. $\neg(E_1 = E_2) \equiv (E_1 \neq E_2)$

2.3.2 Algebraic Laws

Laws of Arithmetic Operators

Algebraic laws for arithmetic operators are defined. Landau describes theories for the arithmetic of numbers [Lan66]. Commutative, associative and distributive laws are defined. The algebra is defined in terms of the following laws:

1. **Commutative Laws:**

$$(x + y) \equiv (y + x)$$
$$(x * y) \equiv (y * x)$$

[5] E_1, E_2 and E_3 are expressions.

2. **Associative Laws:**

$$x + (y + z) \equiv (x + y) + z$$
$$x * (y * z) \equiv (x * y) * z$$

3. **Distributive Laws:**

$$x * (y + z) \equiv (x * y) + (y * z)$$

Laws of Logical Operators

Equivalence laws are defined using the logical operators. They are based on the equivalence laws in [Gri81]. E_1, E_2 and E_3 in the following are expressions.

1. **Commutative Laws:**

$$(E_1 \& E_2) \equiv (E_2 \& E_1)$$
$$(E_1 | E_2) \equiv (E_2 | E_1)$$
$$(E_1 = E_2) \equiv (E_2 = E_1)$$

2. **Associative Laws:**

$$(E_1 \& (E_2 \& E_3)) \equiv ((E_1 \& E_2) \& E_3)$$
$$(E_1 | (E_2 | E_3)) \equiv ((E_1 | E_2) | E_3)$$

3. **Distributive Laws:**

$$(E_1 | (E_2 \& E_3)) \equiv ((E_1 | E_2) \& (E_2 | E_3))$$
$$(E_1 \& (E_2 | E_3)) \equiv ((E_1 \& E_2) | (E_1 \& E_3))$$

4. **De Morgan's Laws:**

$$\neg(E_1 \& E_2) \equiv (\neg E_1 | \neg E_2)$$
$$\neg(E_1 | E_2) \equiv (\neg E_1 \& \neg E_2)$$

5. **Law of Negation:** $\neg(\neg E_1) \equiv E_1$

6. **Law of the Excluded Middle:** $(E_1 | \neg E_1) \equiv TRUE$

7. **Law of Contradiction:** $(E_1 \& \neg E_1) \equiv FALSE$

8. **Laws of Implication:**

$$(\textbf{IF } TRUE \textbf{ THEN } E_1) \equiv E_1$$
$$(\textbf{IF } TRUE \textbf{ THEN } E_1 \textbf{ ELSE } E_2) \equiv E_1$$

$$(\textbf{IF } FALSE \textbf{ THEN } E_1 \textbf{ ELSE } E_2) \equiv E_2$$

9. **Laws of OR-simplification:**

$$(E_1|E_1) \equiv E_1$$
$$(E_1|TRUE) \equiv TRUE$$
$$(E_1|FALSE) \equiv E_1$$
$$(E_1|(E_1\&E_2)) \equiv E_1$$

10. **Laws of AND-simplification:**

$$(E_1\&E_1) \equiv E_1$$
$$(E_1\&TRUE) \equiv E_1$$
$$(E_1\&FALSE) \equiv FALSE$$
$$(E_1\&(E_1|E_2)) \equiv E_1$$

2.3.3 Rewriting Rules

Based on the algebraic laws described in the previous subsection, the rules of substitution and transitivity are defined as follows:[6]

- **Substitution Rule:**

  $e_1 = e_2, P \vdash P[e_1/e_2]$,

  where e_1 and e_2 are expressions, P is a predicate and $P[e_1/e_2]$ is generated by replacing the e_1 in P by the expression e_2.

- **Transitivity Rule:**

  $e_1 = e_2, e_2 = e_3 \vdash e_1 = e_3$,

  where e_1, e_2 and e_3 are expressions.

2.3.4 Induction Rule

We may have an equation, $x(sub) =$ **IF** $sub = 1$ **THEN** 0 **ELSE** $x(sub-1)+1$ and a proof goal, A: $x(sub) = sub - 1$ for $1 \leq sub \leq SIZE.x$. Notice that the proof goal is based on the subscript, *sub*, that is a natural number. We can prove the proof goal by *induction* on *sub*.

First, the basis of the induction, namely $x(1) = 1-1 = 0$, is proven by substitution and the arithmetic laws. As an inductive hypothesis, we assume $x(sub-1) = (sub -$

[6] $\vdash$ is a symbol of derivability. $A \vdash B$ means that a wff B is derived from a wff A by a law of the MODEL calculus.

$1) - 1$, where $2 \leq sub \leq SIZE.x$. According to the equation, $x(sub) = x(sub-1) + 1$. By replacing the variable, $x(sub-1)$, by the expression, $(sub-1)-1$, as assumed in the inductive hypothesis, we have $x(sub) = ((sub-1)-1)+1$. Hence, $x(sub) = sub - 1$. We conclude that the proof goal is proven.

Such a proof technique of induction on its subscript variable is stipulated as follows:

- **Induction Rule:**

  basis: $A(1)$ holds, where A is a proof goal.
  step: If $A(i-1)$ holds, so does $A(i)$, for all $2 \leq i \leq SIZE_i$.
  conclusion: A holds for $1 \leq i \leq SIZE_i$.

2.4 Example

This section demonstrates the methodology of verification based on *equational reasoning*. It only involves substitution and transitivity based on the algebraic laws of equivalences where reflexivity and symmetricity are encoded by the equivalence relation ($\equiv$). For a set of equations in a specification (that are regarded as axioms) and its requirement assertions (axioms), a proof goal (also expressed as an equation) is given. We rewrite the equations in the specification using the rewriting rules and the algebraic laws until we get the proof goal. A proof formulated under the equational reasoning is a sequence of substitutions. This contrast with other formal reasoning methods such as "natural deduction system" which employs a Gentzen style implication rule [Man74, Gri81, Gal86, Lin88]. For our purpose of reasoning for an equational language like MODEL, the equational reasoning is adequate.

Most verification methodologies keep track of changes of execution states that are values of program variables [Man74]. Since a variable has a single value in a mathematical language such as MODEL, we never trace changes of execution states (values of program variables) during the verification of a mathematical language program.

Unlike other mathematical languages such as Lucid [AW76, AW77], no temporal operators (**first**, **next**, **as soon as** etc.) are necessary in MODEL. In its calculus, passage of time [MP81, OL82, Lam83, Kro87] is replaced by the notion of implicit universal and existential quantifiers (the conditions of existence of variables) and "firing" of equations. In some cases, the order of subscripts corresponds to the ordering of firing equations to determine variables. In other cases, "firing" of equations may be in parallel. The calculus provides these notions in place of procedurally representing the notion of time-passing and proving properties related to time-passing.

Usually, programs being tested are written in a programming language that a user is familiar with. Their requirement assertions and proof goals are normally expressed in the *object language*[7] of the formal verification system is often very formal and rigid.

[7] a logical language in which propositions are expressed and reasoned about

It has been pointed out that the user must spend much time and energy translating the "natural" descriptions of the programs, the requirements and the proof goals into the "complicated" logical forms of the object language [Lin88]. Ideally, the object language of the formal verification system should be close to the language of programs. In this methodology, the object language is same as the specification language.

The problem of finding a greatest common divisor (gcd) of two positive integers is chosen as an illustrating example. As discussed in Section 2.1, proving the correctness of the specification requires three inputs: the specification equations, requirement assertions and proof goals. These are discussed in the following. Next, the formal proof of the correctness is presented.

Euclid's algorithm for computing a gcd of two positive integers $x1$ and $x2$ is expressed as a MODEL specification [PLGS88]. Subsequent chapter discusses the use of graphics in composition, modification, testing and verification of MODEL specifications. Therefore, we briefly present here as well this graphical approach. The graphic representation of the gcd example is illustrated in Figure 2.4.

Each instance of an equation in the specification is envisaged as executed as soon as its inputs are available. The execution of the equations may be in parallel. However, there is *precedences* or *dependencies*[8] among the equations and the variables. *Array graph* is a Petri-net like graph that specifies such precedences [Lu81, PP83, SLPP84, SP88].

When a specification is composed, the user conceives variables and equations that are represented in an array graph using icons. The array graph describes variables, equations and precedences among them as shown in Figure 2.4. The simple box icon in the graph denotes a scalar variable, $x1, x2$ and z. Array variables, $y1(i)$ and $y2(i)$, are represented by the icon of a box with lines denoting its dimensionality. The edges expressed as arrows denote the precedences, i.e., data dependencies and parameter precedences. They are labeled by the associated subscript expressions. For example, the edge from the $y1(i)$ node to the `Eq 1` node is labeled by the subscript expression, $i - 1$. It means the value of the element $y1(i - 1)$ is provided as the input of the equation node, `Eq 1`.

As soon as the array graph is completed by the user, a *prompting mechanism* examines the syntax of an array graph, finds LHS variables to be defined, and queries the user to provide the definitions of the LHS variables. For example, a prompt "**Eq1: y1(i) =**" is generated on the the *text window* by the checking mechanism. The user would provide the following equation:

```
IF i=1 THEN IF x1 > x2 THEN x1
                       ELSE x2
       ELSE IF y1(i-1) > y2(i-1) THEN y1(i-1) - y2(i-1)
```

[8]There are *hierarchical precendence* within the structures of the variables, *data dependency* in evaluating the variables of the equations, and *parameter precedence* in evaluating the control variables of the variables.[Lu81, PP83, SP88].

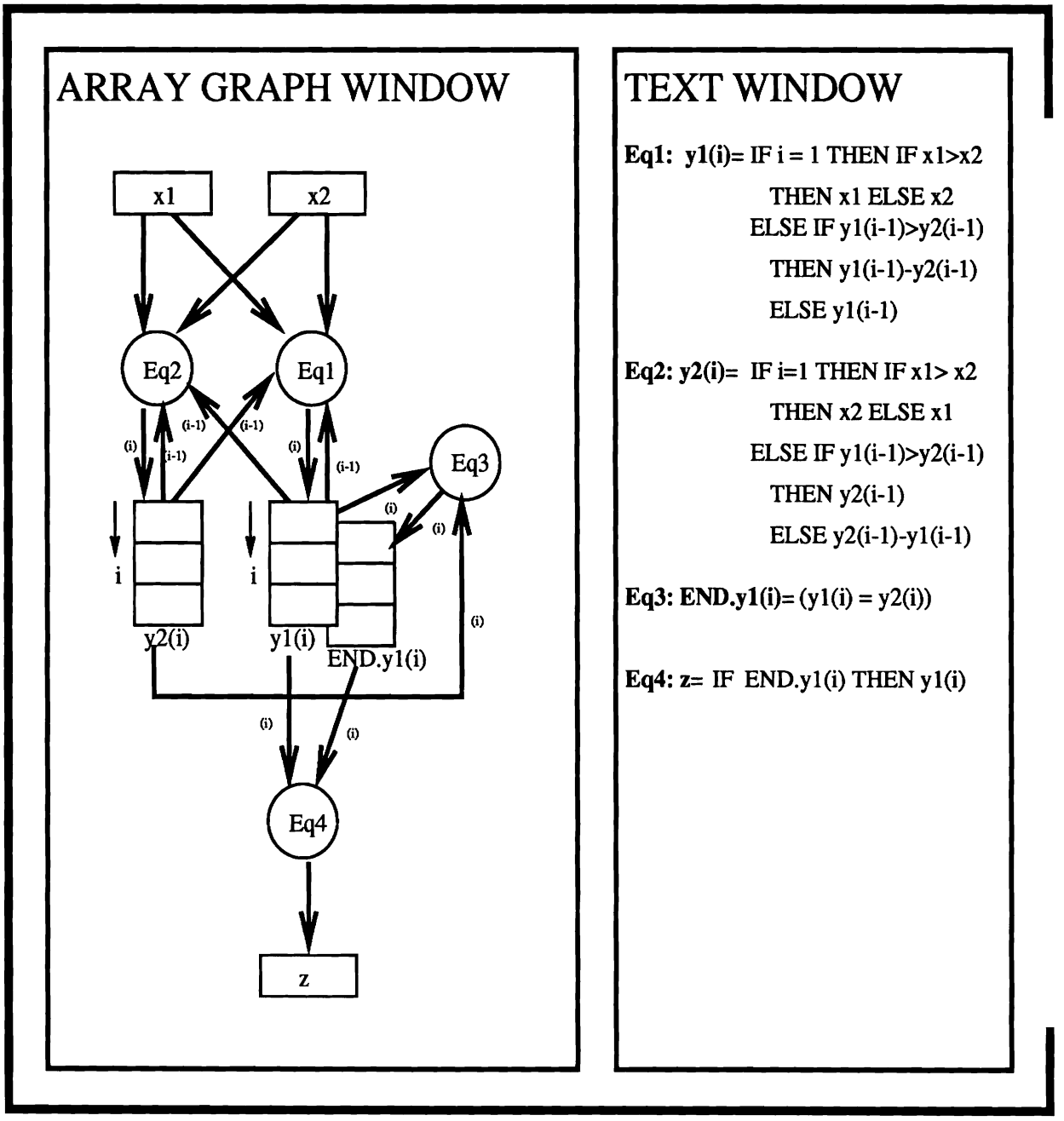


Figure 2.4: A sample MODEL specification computing a GCD of two integer inputs.

```
                                    ELSE y1(i-1);
```

As will be discussed in the next chapter, there is a *checking mechanism* that finds undefined or redundant variables, checks data types and locates inconsistent definitions of subscript expressions and dimensionality of variables. The checking mechanism facilitates the process of formulating an array graph without inconsistent and ambiguous definitions of variables. Inconsistency and ambiguity in definitions of variables are detected and removed by the user through changing the graph or the equations.

Similarly, the query "**Eq 2: y2(i) =** " is generated by the prompting mechanism and the definition of $y2(i)$ is provided as the equation presented in Figure 2.4. Then the checking mechanism is invoked to locate the inconsistency and the ambiguity. `Eq 1` and `Eq 2` in the text window define two array variables $y1$ and $y2$ which contain intermediate values of the computation. The size of the array variables is determined by the control variable $END.y1(i)$. `Eq 3` specifies the expression that determines the values of the elements, $END.y1(i)$'s. The output of the computation, i.e. a gcd of $x1$ and $x2$, is the value of the last element of $y1$. That is, the gcd of $x1$ and $x2$ is $y1(SIZE.y1)$ where $y1(SIZE.y1) = y2(SIZE.y1)$. `Eq 4` defines the operation of determining z, which is $gcd(x1, x2)$.

As discussed in Chapter 1, MODEL is also used in reverse engineering. The graphical user interface can be used to display an existing specification to facilitate understanding and modifying a specification generated from the old procedural programs. In that case, array graph and equations of the specification are displayed on the screen by the system. User tries to understand the meaning of the specification by reviewing the array graph and the equations.

The procedure described above is used in both composing and understanding a MODEL specification through the graphical user interface. The graphics is also used in the stage of interactive verification on the specification. It effectively helps people to understand the internal structures of the specification. The textual representation of equations specifies their operations that cannot be precisely defined through the graphics. The interactive verification process on the specification removes errors at the early stage of the software development. Therefore, the interactive graphical user interface increases productivity of programmers and quality of programs.

Next, the user enters requirement assertions about the specification. They define behavior of the specification and they become axioms in the verification system. The following assertion about a gcd of two positive integers v and w is the basis of the Euclid algorithm:

```
gcd(v,w) =  IF v = w THEN v
                     ELSE IF v < w THEN gcd(v,w-v)
                                   ELSE gcd(v-w,w)
```

Next, the goals for the verification are specified in proof goals. The proof goals consist of the goals and their subgoals as follows:

Goal I. $z = gcd(x1, x2)$

Subgoal 1. $gcd(x1, x2) = gcd(y1(1), y2(1))$

Subgoal 2. $gcd(y1(i), y2(i)) = gcd(y1(i-1), y2(i-1))$

Subgoal 3. $z = gcd(y1(SIZE.y1), y2(SIZE.y1))$

Goal II. $SIZE.y1 < finite\_val$

Subgoal 1. $max(y1(i-1), y2(i-1)) - max(y1(i), y2(i)) \geq 1$

Subgoal 2. $SIZE.y1 < max(x1, x2)$

The first goal is to prove that the specification correctly computes the gcd of the input variables. The second one is to prove that its execution terminates within a finite amount of time. It can be proven by showing that the size of the array $y1$ is bounded by a finite number, i.e. $max(x1, x2)$.

The algebraic laws, the rewriting rules and the induction rule are applied during the process of verification. The subgoals presented in the proof goals are proven one by one. The followings are the steps of the verification. For each step of verification, the names of the rules and/or the algebraic laws used for the verification step are given:

1. Proof of $gcd(x1, x2) = gcd(y1(1), y2(1))$:

   a. $i = 1$, Eq1 $\vdash$
   $y1(1) =$ **IF** $1 = 1$ **THEN IF** $x1 > x2$ **THEN** $x1$ **ELSE** $x2$
   ELSE IF $y1(1-1) > y2(1-1)$ **THEN** $y1(1-1) - y2(1-1)$ **ELSE** $y1(1-1)$;
   (by Substitution)

   b. $y1(1) =$ **IF** $1 = 1$ **THEN IF** $x1 > x2$ **THEN** $x1$ **ELSE** $x2$
   ELSE IF $y1(1-1) > y2(1-1)$ **THEN** $y1(1-1) - y2(1-1)$ **ELSE** $y1(1-1)$;
   $\vdash$
   $y1(1) =$ **IF** $TRUE$ **THEN IF** $x1 > x2$ **THEN** $x1$ **ELSE** $x2$
   ELSE IF $y1(1-1) > y2(1-1)$ **THEN** $y1(1-1) - y2(1-1)$ **ELSE** $y1(1-1)$;
   (by Semantics of Relational Operators)

   c. $y1(1) =$ **IF** $TRUE$ **THEN IF** $x1 > x2$ **THEN** $x1$ **ELSE** $x2$
   ELSE IF $y1(1-1) > y2(1-1)$ **THEN** $y1(1-1) - y2(1-1)$ **ELSE** $y1(1-1)$;
   $\vdash$
   $y1(1) =$ **IF** $x1 > x2$ **THEN** $x1$ **ELSE** $x2$
   (by Law of Implication)

   d. $i = 1$, Eq 2 $\vdash y2(1) =$ **IF** $x1 > x2$ **THEN** $x2$ **ELSE** $x1$
   (by Substitution, Semantics of Relational Operators, and Law of Implication)

CASE $x1 > x2$:

e. (c) $\vdash y1(1) = x1$

f. (d) $\vdash y2(1) = x2$

g. (e),(f) $\vdash gcd(x1, x2) = gcd(y1(1), y2(1))$
(by Substitution)

CASE $x1 \leq x2$:

h. (c) $\vdash y1(1) = x2$

i. (d) $\vdash y2(1) = x1$

j. (h),(i) $\vdash gcd(x1, x2) = gcd(y2(1), y1(1))$
(by Substitution)

k. (j), the requirement assertion $\vdash gcd(y2(1), y1(1)) = gcd(y1(1), y2(1))$

l. (j),(k) $\vdash gcd(x1, x2) = gcd(y1(1), y2(1))$
(by Transitivity)

2. Proof of $gcd(y1(i), y2(i)) = gcd(y1(i-1), y2(i-1))$:

CASE $y1(i-1) > y2(i-1)$:

a. $v = y1(i-1)$, $w = y2(i-1)$, $y1(i-1) > y2(i-1) \vdash v > w$.
(by Substitution)

b. $v > w$, requirement assertion $\vdash gcd(v, w) = gcd(v - w, w)$
(by Law of Implication)

c. $v = y1(i-1)$, $w = y2(i-1)$, (b) $\vdash gcd(y1(i-1), y2(i-1)) = gcd(y1(i-1) - y2(i-1), y2(i-1))$
(by Substitution)

d. $y1(i-1) > y2(i-1)$, Eq 1 $\vdash y1(i) = y1(i-1) - y2(i-1)$
(by Law of Implication)

e. $y1(i-1) > y2(i-1)$, Eq 2 $\vdash y2(i) = y2(i-1)$
(by Law of Implication)

f. (d),(e), (c) $\vdash gcd(y1(i-1), y2(i-1)) = gcd(y1(i), y2(i))$
(by Substitution)

g. $gcd(y1(i-1), y2(i-1)) = gcd(y1(i), y2(i)) \vdash gcd(y1(i), y2(i)) = gcd(y1(i-1), y2(i-1))$
(by Law of Equality)

CASE $y1(i-1) \leq y2(i-1)$:
Likewise, $gcd(y1(i), y2(i)) = gcd(y1(i-1), y2(i-1))$

3. Proof of $z = gcd(y1(SIZE.y1), y2(SIZE.y1))$:

a. $i = SIZE.y1$, Eq 3 $\vdash END.y1(SIZE.y1) = (y1(SIZE.y1) = y2(SIZE.y1))$
(by Substitution)

b. $END.y1(SIZE.y1) = (y1(SIZE.y1) = y2(SIZE.y1)) \vdash (y1(SIZE.y1) = y2(SIZE.y1)) = TRUE$
(by The Rule of Control Variables)

c. (a),(b) $\vdash END.y1(SIZE.y1) = TRUE$
(by Transitivity)

d. $i = SIZE.y1$, Eq 4 $\vdash z =$ **IF** $END.y1(SIZE.y1)$ **THEN** $y1(SIZE.y1)$
(by Substitution)

e. (c),(d) $\vdash z = y1(SIZE.y1)$
(by Law of Implication)

f. Let $v = y1(SIZE.y1)$ and $w = y2(SIZE.y1)$.
$v = w$, requirement assertion $\vdash gcd(v, w) = v$
(by Law of Implication)

g. $v = y1(SIZE.y1), w = y2(SIZE.y1)$, (f) $\vdash gcd(y1(SIZE.y1), y2(SIZE.y1)) = y1(SIZE.y1)$
(by Substitution)

h. (e),(g) $\vdash z = gcd(y1(SIZE.y1), y2(SIZE.y1))$
(by Transitivity)

$\Rightarrow$ Conclude $z = gcd(x1, x2)$ by Induction Rule.

4. Proof of $max(y1(i-1), y2(i-1)) - max(y1(i), y2(i)) \geq 1$:

a. $x1 > 0$, $x2 > 0 \vdash (y1(i) > 0) \& (y2(i) > 0)$
(by Induction Rule)

b. (a) $\vdash (y1(i-1) - y1(i) > 0) | (y2(i-1) - y2(i) > 0)$
(by Induction Rule)

c. (b) $\vdash max(y1(i-1) - y1(i), y2(i-1) - y2(i)) \geq 1$
(by Induction Rule)

5. Proof of $SIZE.y1 < max(x1, x2)$:

a. (4) $\vdash SIZE.y1 < max(x1, x2)$

$\Rightarrow$ Conclude $SIZE.y1 < finite\_val$ by the assumption such that $x1 > 0$ and $x2 > 0$.

Chapter 3

VISUALIZATION

3.1 Introduction

The visualization system employs program visualization, visual programming, pull-down menus, and texts. The objective of this chapter is to describe the visual programming (VP) environment for MODEL. Visual programming allows a user to compose a program in a two dimensional fashion [Har88, Mye88, Cha90]. The VP environment offers an interactive graphical user interface where a user can effectively compose and modify a MODEL specification. It also facilitates generating reliable programs by offering three types of utilities: checking unique computability of variables, testing, and verification. The interactive graphical user interface improves interactions between the user and the environment in composing, modifying, checking, testing and verifying a specification. This chapter describes the VP environment. Its use in checking, testing and verification is described in respective chapters.

There are a number of advantages of visual programming over conventional (textual) programming. A human being can recognize and process multi-dimensional data such as pictures and diagrams. A conventional computer program written in a textual form does not fully utilize the capability of the human for visual information processing. Many researchers have demonstrated that two-dimensional displays of programs are helpful for humans in composing and understanding programs, especially for non-programmers or novice programmers [Mye88, Cha90]. A graphical description can explain a complex problem, such as a concurrent process or a real-time system. Even a professional programmer could benefit from use of high-level graphical descriptions when dealing with very complex problems.

The VP environment for MODEL overcomes some problems with existing VP systems. First, it does not use a control flow diagram, e.g. a flow chart, on which many VP systems are based. It has been proven that a flow chart is a poor abstraction of software structure and useless as a design tool [Bro87]. In the VP environment, an equational language, MODEL, is visualized as a Petri-net like data flow diagram, *array*

graph. Recall that a MODEL specification is a collection of equations and the order of executing them is not specified; every variable has a single value; every equation is fired as soon as its inputs are available. As a result, the user does not have to trace flows of program states around the whole diagram. He can examine properties of any MODEL equation or a group of connected equations locally in the displayed graph.

A MODEL equation is regarded as residing on a processing element of a data flow machine. It is fired as soon as all of its inputs are available. Each equation is represented as a node. Each variable (whether a scalar or a multi-dimensional array variable) also forms a node. A hierarchical structure of a variable is denoted by a set of variable nodes and edges of *hierarchical precedences.* The other types of edges visualize *data dependency* and *parameter precedence* among equations and the input and output variables. The array graph has more expressive power and is more readable than a flow chart.

The VP environment combines graphics and an equational language, MODEL. A data flow graph of a specification (= array graph) is represented in graphics to help the user understand its meaning. At the same time, compact and intuitive mathematical definitions of the equations are precisely expressed. Both the graphics and the mathematics complementarily enhance the user's understanding of the specification.

As a specification is composed, its consistent definitions/references (of the dimensionality of array variables, the ranges of dimensions, the data dependency, etc.) is interactively checked as discussed in Chapter 4. The consistency of the specification is also tested after its composition and/or its modification. As will be discussed in the following chapter, the *testing*[1] of a specification may be performed [Kin76, Cla76, DLS78, RW85, How87, Ham88, GH88, Bei90]. The graphical user interface facilitates the testing procedure. The correctness of a specification may be proven using the verification system through the interactive graphical user interface. Note that the procedures of checking, testing and verification of programs are frequently carried out by users incompletely, partially because of their complexity and tediousness. The combination of interactive checking, testing, verification and graphics will make those procedures easier.

There are a number of graphic systems that can be utilized for the graphical user interface[Dec90, BMSW90]. DECdesign is a software development environment based on graphics [Dec90]. It helps a user to analyze and design software systems according to sound rules of the design methodology. As an example, the design methodology of the Yourdon Data Flow Diagram (DFD) was implemented in DECdesign. It provides an *icon-based graphical editor* where a user creates a diagram using icons. The icons and their meanings may be based on the Yourdon DFD notation. A graph has two edit windows: the *graphics window* and the *forms window.* The graphics window displays the diagram. To edit or examine information about a graph, a user opens the respective forms window.

[1] a process of discovering faults that cause failure of software

The VP environment for MODEL is proposed to be implemented in terms of any one of existing graphic systems. We propose to customize DECdesign by providing the methodology of visual programming to the DECdesign *core environment* in terms of graphical objects and rules. An array graph is displayed on the graphical window for the VP environment (called *array graph window*). Its data declarations and equations are presented in other window (called *text window*).

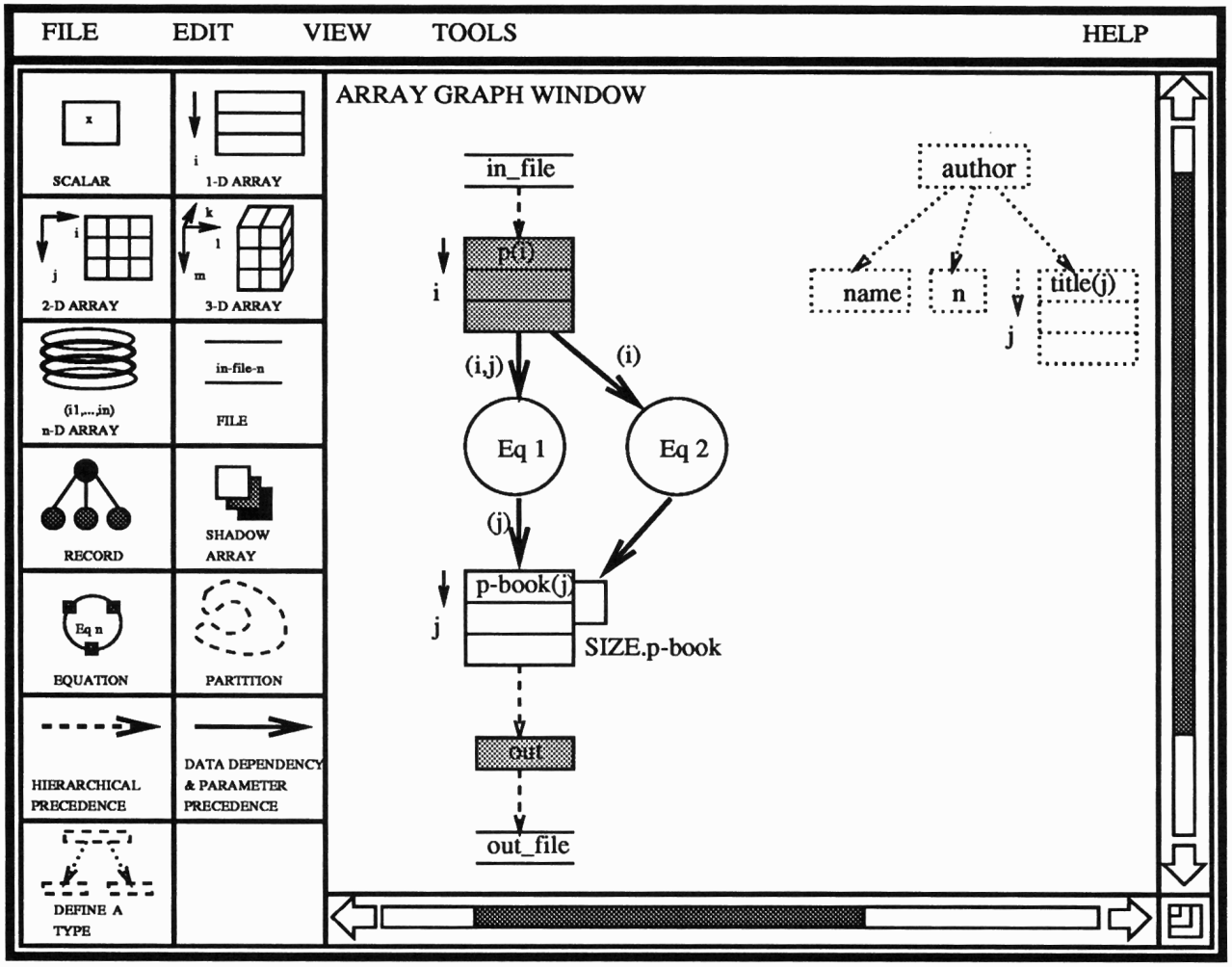
This chapter discusses mainly the composition and modification of specifications using the VP environment. Section 3.2 illustrates the idea of the visual programming. The graphical user interface and its use in the composition and the modification are described in Section 3.3.

3.2 Example

A user composes a specification by constructing an array graph in graphics. Declaring data structures and defining equations in MODEL are performed textually. The variables and the equations of the specification are represented by graphical objects (icons) in the array graph. The mouse and the keyboard inputs are used in formulating and modifying an array graph. It is similar to an *icon-based graphical editor* [TB86].

As illustrated in Figure 3.1, a *view* of an array graph is seen through the *array graph window*. It consists of a "canvas" for drawing the graphs, a menu of icons and pull-down menus ("FILE", "EDIT", "VIEW", "TOOLS" and "HELP"). The graph denotes the data structures, the equations and the precedence of executing the equations graphically. Each equation has its mathematical definition in the corresponding *text window*. The VP environment facilitates the maintenance of consistency in the contents of the windows.

For example, consider a simple MODEL specification of information retrieval in Figure 3.1. The module, `info_retrieval`, finds a list of books written by "Prywes" from the input file, `in_file`, where the titles of the books are stored under the names of their authors. The equation, `Eq 1`, copies titles of the books written by "Prywes" to the 1-D array, `p_book(j)`. The size of the array is limited by the control variable, `SIZE.p-book`, determined by the equation `Eq 2`. The array, `p_book(j)`, forms a record, `out`, of the target file, `out_file`. The input data are the records, `p(i)`, of the data type, `author`. It consists of author's name (`name`), an array of titles (`title(j)`) and the size of the array (`n`). The structure of the data type, `author`, is graphically expressed in the array graph window (in terms of icons with dotted lines) while its textual declaration ("TYPES:") and its usage ("VARIABLES:") are presented in the text window.



TEXT WINDOW

HEADER:

MODULE: info_retrieval;
SOURCE: in_file;
TARGET: out_file;

DATA DECLARATIONS:

TYPES:

1 author IS GROUP,
2 name IS FIELD (CHAR(20)),
2 n IS FIELD (NUM),
2 title(j) IS FIELD (CHAR(80));

VARIABLES:

1 in_file IS FILE,
2 p(i) IS RECORD TYPE author;

1 out_file IS FILE,
2 out IS RECORD,
3 p_book(j) IS FIELD (CHAR(80));

EQUATIONS:

Eq1: p-book(j) =
IF (p(i).name = "Prywes") THEN p(i).title(j);

Eq2: SIZE.p-book =
IF(p(i).name = "Prywes") THEN p(i).n;

Figure 3.1: An array graph window and its text window

3.3 Graphical User Interface

The graphical user interface of the VP environment manages multiple windows on a screen, invokes utilities requested by a user, accepts keyboard and mouse input, and displays an array graph of a MODEL specification, equations, and messages.

The utilities of managing multiple views and windows are described in Section 3.3.1. The graphical objects such as icons and menus are discussed in Sections 3.3.2 and 3.3.3. The methodology of composing a specification is explained in Section 3.3.4. The graphical user interface also facilitates modifying a specification graphically and textually. The modification procedure is presented in Section 3.3.5. The graphical operations on the displayed array graphs are explained in Section 3.3.6.

3.3.1 Views and Windows

A user can selectively have many different *views* of an array graph of a specification. For instance, the user may want to see only field variables, equations and data dependencies of the array graph. Or he may want to modify the graph through graphical operations such as *zooming* a part of the graph, *imploding* a partition of the graph, *exploding* imploded partitions, etc. A graphical representation of either an array graph itself or a result of the graphical operations on the graph is called a *view* of the array graph.

The user can "see" the view through a *window*. The physical size of the window may or may not cover the whole view. In case that the window is not big enough to show the whole view, the user may move the window around the view to get the whole picture. Figure 3.2 illustrates the relationship among an array graph, a view and a window. An array graph of a specification is displayed in Figure 3.2-(a). A user may delete all field nodes of the input and the output of the specification, `x1, x2, x3, x4, x11` and `x12`, to replace them by their ancestor nodes, `in_grp` and `out_grp`, as shown in Figure 3.2-(b). The user creates the view. The window of the VP environment may be able to cover only a portion of the view. In Figure 3.2-(b), the user can "see" the equations, `Eq 1, Eq 2, Eq 3, Eq 5` and `Eq 8`, the variables, `x5, x6` and `x7`, and the data dependencies among them through the window.

A *scrolling* operation is needed when the user moves the array graph window around a view. As shown in Figure 3.1, the array graph window has two *scroll bars* for moving the window (up-and-down and left-and-right). The size of the array graph window can be changed using the *resize* icon in the lower-right corner of the window. He can also enlarge a part of the graph by *zooming*. When the user examines a complicated graph, the zooming operation is useful. These graphical operations are already implemented as standard ones in X window system [SGN88, AS90] and DECdesign [Dec90].

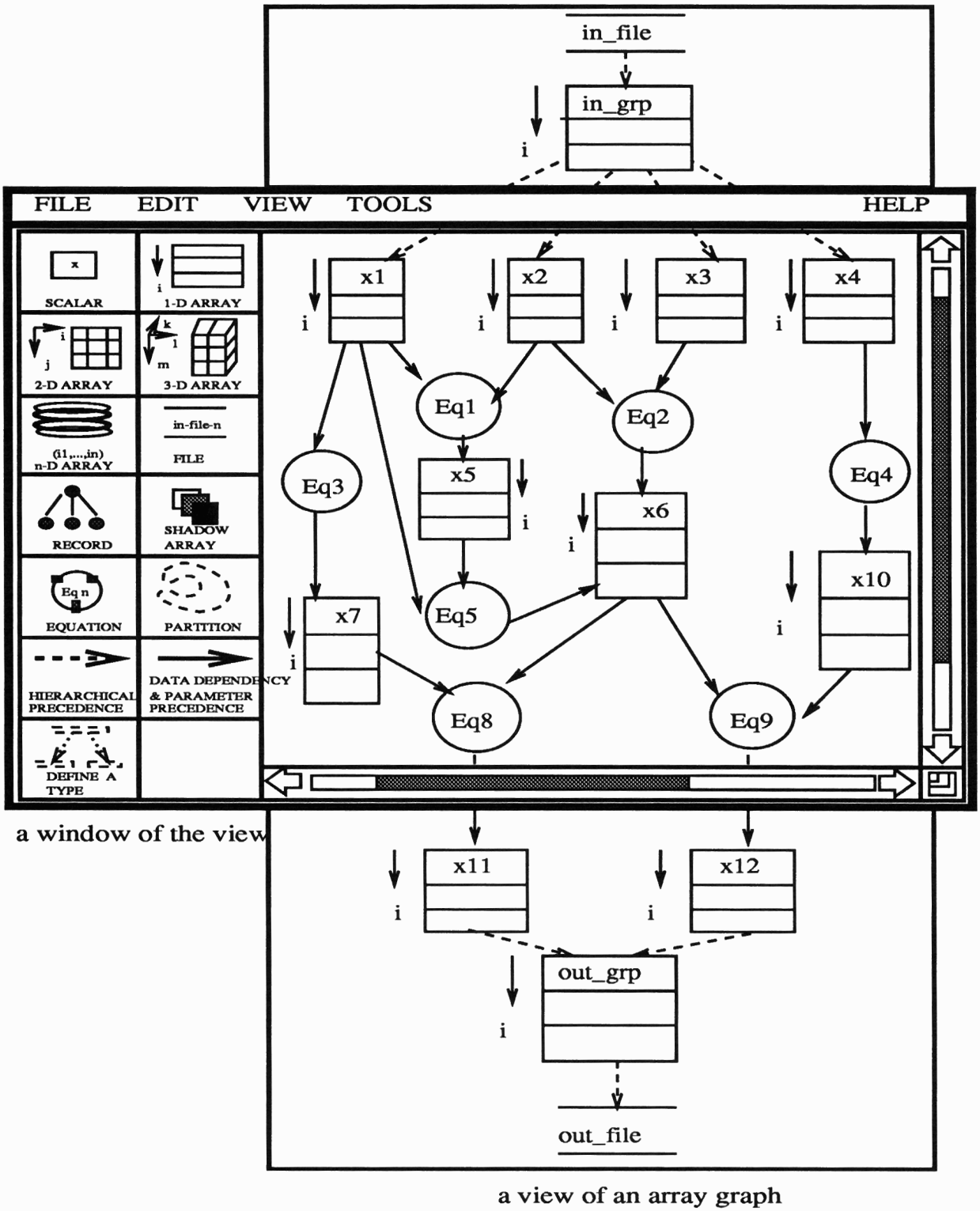


Figure 3.2: An array graph, a view and a window.

3.3.2 Graphical Objects

An array graph consists of variables, equations and their precedences. The variables and the equations are *nodes* of the graph, while the precedences are denoted by *edges* of the graph. The nodes and the edges are created by selecting the icons illustrated in Figure 3.1.

A scalar variable is expressed as a box icon. Multi-dimensional array variables (1-D, 2-D, 3-D and n-D arrays) are denoted by boxes and subscripts as presented in Figure 3.1. Those icons of variables are called *data icons.*

The hierarchical structure of a variable is also expressed by the data icons, the "FILE" and the "HIERARCHICAL PRECEDENCE" icons in Figure 3.1. For example, consider the hierarchical structures for `in_file` and `out_file` declared in the text window of Figure 3.1. The variables, `name, n, title(j)` and `p_book(j)`, are called *fields.*[2] The fields may form a logical unit called a *group*[3] or a physical *record.*[4] A group and a record could be regarded as a scalar variable like `out` or a multi-dimensional array variable like `p(i)`.

We can distinguish a field node from a group node by hierarchical structure in an array graph. However, a record node is not distinguished from a group node. The "RECORD" icon is provided to mark a record node by shading. A user first clicks the "RECORD" icon and selects a proper data icon for a record node. Then the selected icon is shaded and appears on the array graph window. The `p(i), out` and `author` nodes in Figure 3.1 are shaded to denote that they are records.

The files,[5] `in_file` and `out_file`, may have a number of groups or records. They are denoted by the "FILE" icon. The "SOURCE" (= input) and the "TARGET" (= output) files of the module are declared in the header part.

Control variables such as *END*, *ENDFILE* and *SUBLINEAR* [MOD89] are represented by the "SHADOW ARRAY" icon which is attached to the corresponding arrays. For example, the 1-D array, `p(i)`, in Figure 3.1 has a shadow variable, `ENDFILE.p(i)`. It has the same shape (dimensionality and range) of the record, `p(i)`. The shadow array icon is attached to the 1-D array icon of `p(i)` as its shadow. With the notion of the shadow array icon, we can reduce the complexity of the array graph and can graphically denote the relationship between an array variable and its shadow array.

An equation is denoted by the circle icon. Inside the circle, the name of the equation such as `Eq 1` is specified. Each equation has a number of inputs and a single output. They are expressed as *attach points* on the icon. For instance, the circle icon in Figure 3.1 has three attach points. Two of them are its inputs and the rest is its output. The

[2] leaf nodes of the tree that shows the entire hierarchy of the declared data structure

[3] a non-leaf node of the tree

[4] a physical unit of communicating with external devices

[5] a root of the tree

attach points disappear as soon as they are connected to edges from all inputs and output of the equation.

As will be discussed in Section 3.3.6, a user may *partition* a class of equations that can be classified as a single equation. A collection of data structures can be partitioned to form a new hierarchical data structure. It is used to simplify the displayed array graph. A user selects the "PARTITION" icon from the icon menu in Figure 3.1 and surrounds nodes that form a new node.

A user may define a new data type. He is able to create a new data type using the data icons and the hierarchical precedence icons. It is implemented by the icon of "DEFINE A TYPE". For example, the data type, `author`, in Figure 3.1 is created using the icon. The 1-D array, `p(i)`, is defined as an instance of the type as presented in the text window in Figure 3.1. To distinguish the declaration of a new data type from normal data declaration, all icons used in the new type declaration are expressed by dotted lines. A generic equation can be defined as a new type equation using the "DEFINE A TYPE" icon. The user clicks the icon and defines a new type equation in terms of variables and equations. Both a new type variable and a new type equation are instantiated by selecting their definitions on the window.

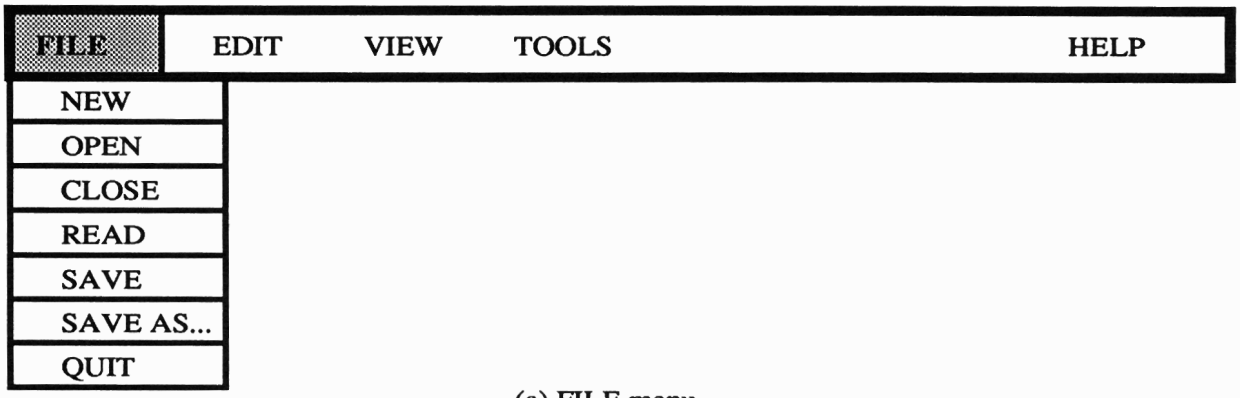
The text window contains textual representations of equations, data declarations and user-defined data types for the graphical objects on the array graph window. It gives accurate and detailed information of the array graph window. The graphical objects on the array graph window and their textual representation must be consistent. The user and the VP environment can eliminate discrepancies between the two different representations (graphics and text) using the checking mechanism. The two different representation of a same specification complementarily helps the user's understanding of the specification.

3.3.3 Pull-Down Menu

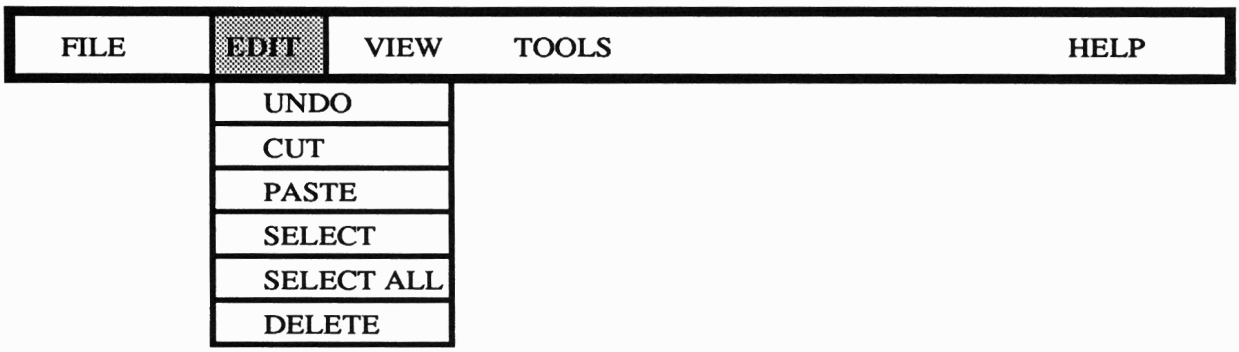
Some operations are invoked using the pull-down menus. File operations such as creating a new array graph, reading a whole or a part of an existing array graph etc. are invoked by choosing the "FILE" menu in Figure 3.3-(a). A user can edit graphical objects on the graph by choosing the edit operations listed under the "EDIT" menu in Figure 3.3-(b).

There may be multiple views of an array graph. The user may want to save the generated views as a report. The graphical user interface offers a utility of making a *snap-shot* of a currently displayed array graph. The "VIEW" menu has the selections of graphical operations as shown in Figure 3.3-(c).

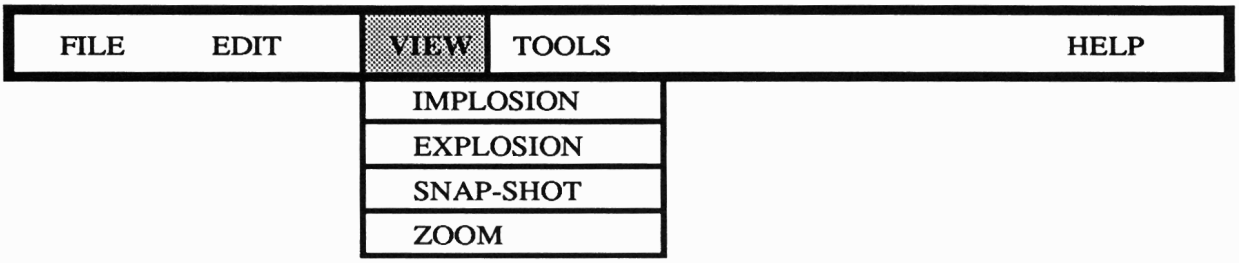
Utilities for composing, checking, testing and verifying a MODEL specification are categorized as "TOOLS". The user can initiate the operations by choosing the menu of Figure 3.3-(d). Finally, the "HELP" menu provides an on-line help facility.



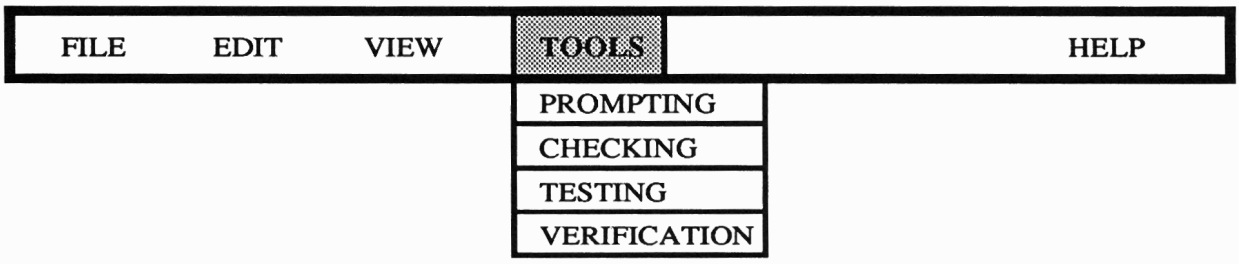
(a) FILE menu



(b) EDIT menu



(c) VIEW menu



(d) TOOLS menu

Figure 3.3: The "pull-down" menus

3.3.4 Composition

A user composes a specification by creating nodes of its variables and equations and connecting them using edges of precedences. For the input and the output variables, the hierarchical structures (declarations) must be specified in both a graphical form and a textual form. The data icons are used to define fields, records, groups and files. A record node is shaded. Their hierarchical precedences are denoted by the broken arrows. A new data type can be created by the user. Its graphical representation in dotted lines appears in the array graph window. The textual declaration of the data type is presented in the text window. The equations are declared as the circle icons on the array graph window. The data dependencies and parameter precedences among the variables and the equations are denoted by the edges of the solid arrow icon. The edges must be annotated by the subscript expressions that are used in the corresponding equations.

Once the graph is completed, the user invokes the *prompting mechanism* from the "TOOLS" menu to get the textual definitions of the equations via the text window. The prompting mechanism recognizes the LHS variables and their corresponding RHS expressions. The mechanism issues a prompt such as "**Eq 1: x1(i) =**" meaning to query the user the textual definition of the RHS expression for the LHS variable, `x1(i)`. The RHS expression should be able to define the value of the LHS variable uniquely as long as its existence condition is satisfied.

As discussed in Chapter 2 and 4, the user must meet the requirements (the existence condition, the computability of equations, etc.) when he composes a specification. The user can detect ambiguous definitions and incomplete definitions of variables and equations as completing the array graph using the prompting mechanism. Then, he starts the *checking mechanism* using the "TOOLS" menu. It checks the consistent definitions and references of variables and equations in the specification. The user interactively composes and checks a specification using the VP environment.

3.3.5 Modification

A user can modify a specification by changing its array graph and/or its data declarations and equations.

The user can *rearrange* the positions of icons (nodes and edges) of an array graph on the array graph window. It is a graphical change of the graph and does not affect the meaning of the array graph and the specification.

A new variable and its hierarchical structure can be *created* and *inserted* into an array graph. By selecting a data icon from the menu, the user can create a new variable node of an array graph. The new variable can be an input of an equation but it should not be an output of an existing equation. If the variable is created as an input of an equation, the user must change the definition of the corresponding equation on the text

window. The data dependency or the parameter precedence relationship between the new variable and the equation is also specified by connecting them with the arrow icon. The dimensions, subscript expressions and their ranges must be consistently specified along the edge.

In case of creating a new equation, the user modifies both the array graph window and the text window. A circle icon denoting an equation is newly inserted into the array graph. Next, the output variable node and its hierarchical structures are defined. The data dependency or the parameter precedence between the equation and its output must be specified in terms of the solid arrow icons. The input variables of the new equation may already exist or must be created. The creation of its new input variables is as same as discussed before. The data dependencies or the parameter precedences between the input variables and the equation are represented by the arrow icons. Finally, a textual expression of the new equation is specified via the text window by invoking the prompting mechanism. The incomplete definitions, the ambiguous definitions and any inconsistent definitions and references among the variables and the equations must be checked and removed. To perform these jobs, the prompting and the checking mechanisms are invoked during the modification procedure.

3.3.6 Graphical Operations

A complicated array graph may obstruct user's understanding of the meaning of a specification. Thus a user simplifies a complicated graph by *imploding* a class of equations into a single equation node. Accordingly, a cluster of variables are imploded into a single variable node. For example, the equations and the variables of the array graph of Figure 3.4 can be *partitioned* (by selecting the "PARTITION" icon) according to their functional behaviors. The partitioned node must behave like a single node. The following rules are stipulated for partitioning:

(1) A *closure*[6] can be partitioned to form an imploded equation node.

(2) A set of equation nodes can be an imploded equation node, if the resulting node can have a single output variable node.

(3) A set of variable nodes can be an imploded variable node, if the resulting node can be defined by a single equation node.

For example, the two closures in Figure 3.4 can be partitioned and named `Part-Eq 1` and `Part-Eq 2`, respectively. The set of equations, {`Eq 9, Eq 10, Eq 11`}, can be partitioned to make an imploded equation node, `Part-Eq 3`, since the set of variables, {`output 1, output 2, output 3`}, can form an imploded variable node and vice versa. A simplified version of the array graph is presented in Figure 3.5.

[6]It is a set of equations and variables that can be executed as one loop [Lu81]. A closure has a single output variable. The equations and the variables in a closure share the same subscript of the same range.

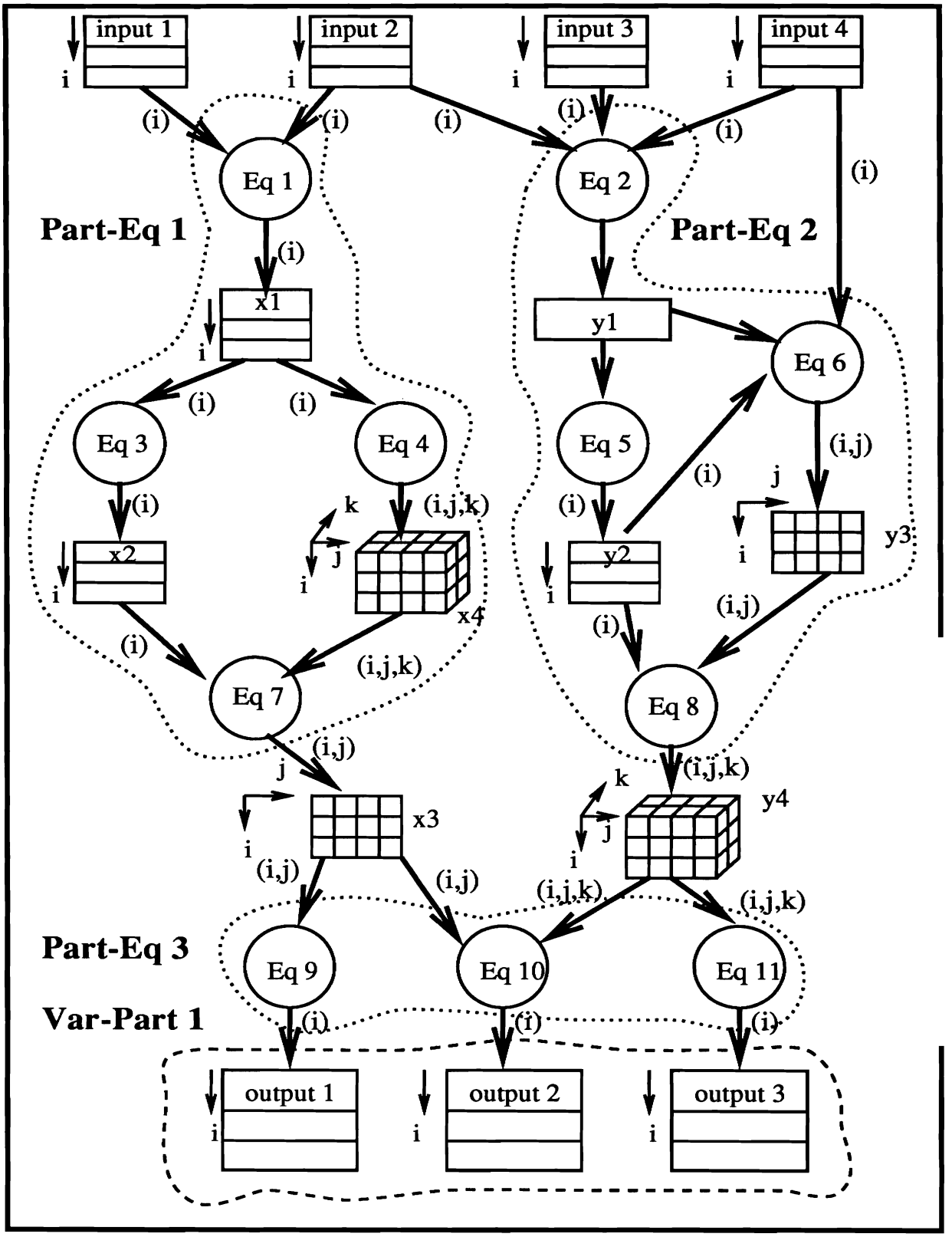


Figure 3.4: Partitioning equations and variables.

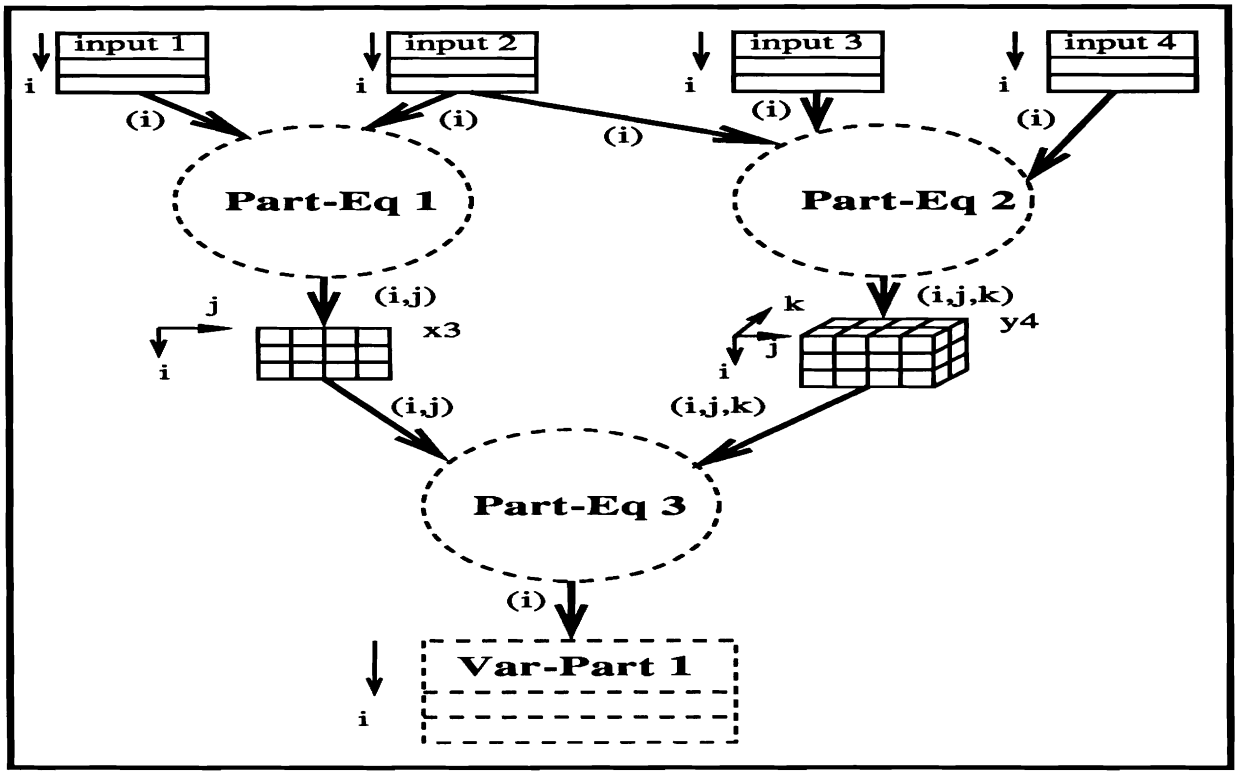


Figure 3.5: An "imploded" array graph.

Notice that the imploded equation node, `Part-Eq 1` in Figure 3.5, does not have the subscript, `k`, which denotes the third dimension of the variable, **x4**, in Figure 3.4. Since the subscript, `k`, is internal in the closure, it does not appear in its imploded node.

On the other hand, a user may want to examine the details of nodes of the simplified array graph. The user can *explode* any imploded node by creating another window. For example, the imploded equation node named `Part-Eq 3` of Figure 3.5 is created by merging three equations, `Eq 9`, `Eq 10`, and `Eq 11`. The explosion operation reveals the detailed structure of the node and its input nodes and its output node as shown in Figure 3.6. Similarly, the imploded variable node can be exploded and its structure and its corresponding equation nodes are exposed as shown in Figure 3.6.

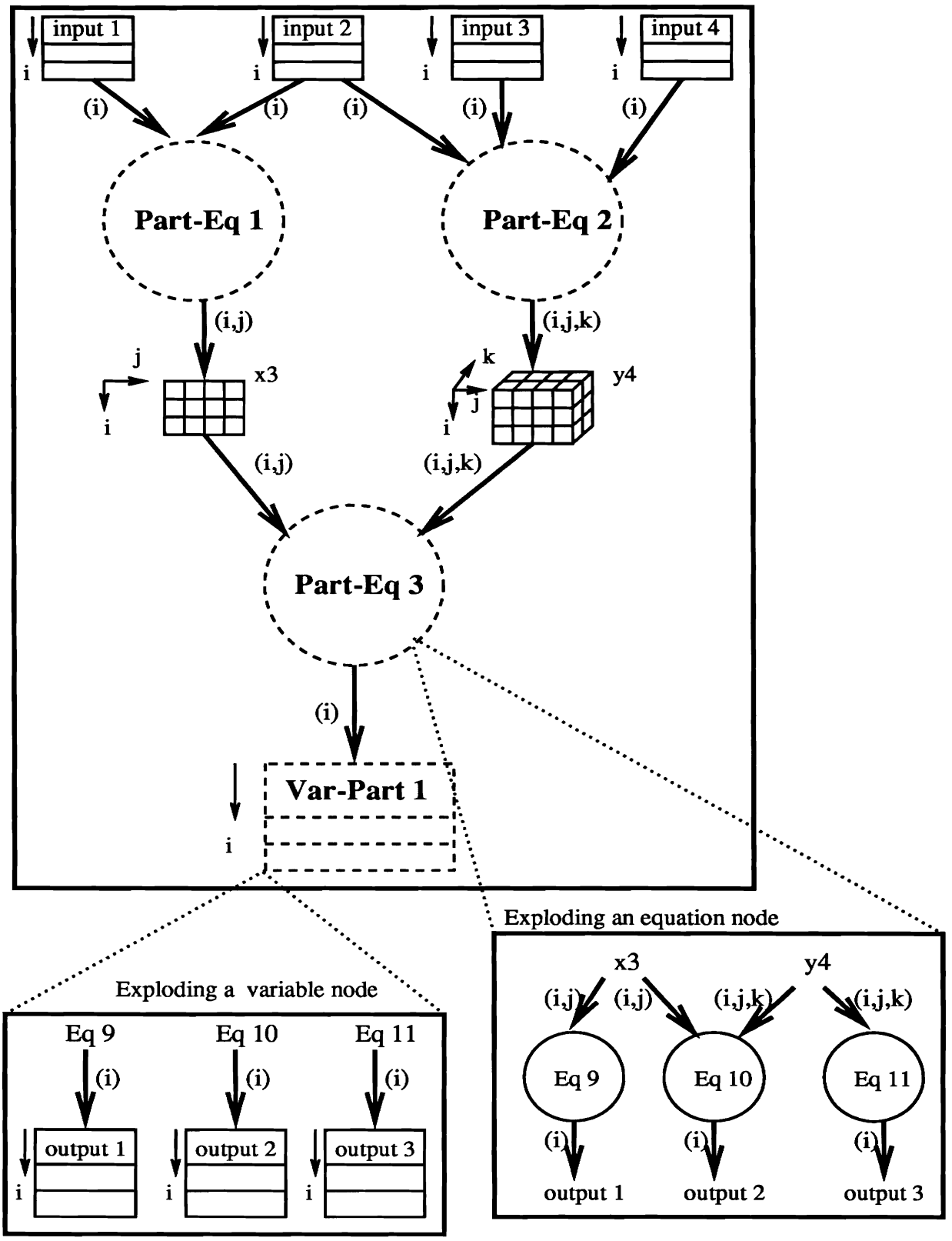


Figure 3.6: The imploded nodes of the array graph are exploded.

Chapter 4

CHECKING

4.1 Introduction

This chapter describes the checking system of MODEL. As discussed in Chapter 2, a user must compose a MODEL specification adhering to a number of requirements. The most important requirement is the existence of all variables of the equations and at least one *causal chain* from input to output[GR89] via the equations of a specification. Thus a unique output (= a solution for the equations) must be always computable for a given input set. A corollary requirement is that its execution must terminate.

The checking system helps a user in composing a specification that complies with these requirements. The checking system performs *static checking* that detects errors or warns the user when satisfying these requirements is conditioned on the values of input data. Responding to the error messages, the user fixes the faults in the specification. He either modifies the specification or sets up assertions that specify the condition of successfully computing equations, regarding to the warning messages. For example, consider an equation, "$x(i)$ = `IF` $a(i) > b(i)$ `THEN` $y(i)$". It cannot define the values of $x(i)$ if the condition, $a(i) > b(i)$, is never satisfied for all i. The checking system issues a warning message to the user on the equation. It is required that he must either modify the equation or provide an assertion that assures the satisfiability of the condition such as "$\exists i, a(i) > b(i)$". Such an assertion becomes a part of the axioms of the MODEL proof system when the correctness of the specification is verified.

The checking system aims to check the computability of the equations. That is, the checking system facilitates a user composing a specification which has one output set (= a solution) of the specification (= a set of equations) for a given input set. The resulting specification consists of the checked equations and the generated assertions that specify the requirements of the computability. They become axioms of the proof system. Since every variable of the equations has a unique definition, the axioms transformed from the equations cannot contradict each other. It follows that the set of axioms are consistent. That is, the checking mechanism facilitates the user in generating the axioms consistent.

The inconsistency of the axioms makes the soundness and the completeness of the proof system, which consists of the axioms and the rewriting rules based on the equivalence laws, meaningless. In that sense, the checking system supports the soundness and the completeness of the proof system.

(1) Ambiguous definitions and incomplete definitions of the variables and the equations are detected as their array graph and *dictionary* [Lu81, MOD89] are completed.[1] If the same name is used for more than one data structures, it causes ambiguity. It is resolved by using *qualifying* (or *prefixing*) names for variables [Lu81, SLPP84]. The incomplete definition of an LHS variable of an equation means that it does not have an RHS expression that defines its value. Also an interim variable may not be explicitly declared. Section 4.2 describes a method of discovering the ambiguous and the incomplete definitions through completing an array graph.

(2) The existence requirement for defining variables is checked. It is examined by checking the definitions of variables and their references. There may be discrepancies between declarations and references about dimensionalities, data types of variables and ranges of subscript expressions. They are checked by *propagating* attributes such as *dimensions* and *ranges*[2] via edges of an array graph. The existence requirement checking is discussed in Section 4.3.

(3) The presence of a causal chain that computes a solution set for a given input values is checked. A cyclic definition of variables (called *circular logic* [SLPP84]) causes an infinite computation. It should not be a part of any causal chain. The checking mechanism detects such a cycle in an array graph and tests if it results a cyclic definition. The method of detecting and removing the cyclic definitions of variables is presented in Section 4.4.

(4) The condition of terminating the execution is checked for the specification. Even though there is no circular logic in an array graph, it may not terminate if their control variables specifying the termination condition such as END do not have finite values. The presence of such control variables and their computability are checked. If they are not found on a causal chain or their values cannot be determined as finite, a warning message is generated. Checking the termination condition is discussed in Section 4.5.

4.2 Graph Construction

Every variable and its definition are listed in the dictionary of a MODEL specification. An array graph is a graphical implementation of a dictionary. A user invokes the prompting mechanism to complete an array graph. The mechanism recognizes every

[1] The dictionary is an internal representation of a specification. The header, the detailed declaration of variables, the equations and their precedence relationships are stored in the dictionary.

[2] size of a dimension of an array variable.

variable and the corresponding equation in the array graph displayed on the array graph window. A textual definition of the equation is requested by the prompting mechanism and provided the user via the text window. The two representations (graphical and textual) on the windows must be consistent. If ambiguous definitions and incomplete definitions of variables are found, an error or a warning message is issued.

4.2.1 Ambiguous Definitions

When several data structures have the same name, it is ambiguous to reference the data structures from equations. The ambiguity is removed as an array graph is constructed. It is done by applying the following rules [Lu81, SLPP84]:

(1) An LHS may reference only interim or output variables.

(2) An RHS may reference also input variables.

In many cases, however, it is necessary to require the user to remove the ambiguity. He may rename the variables by *qualifying* (or *prefixing*) them [SLPP84]. For example, we may have the following declaration statements:

```
1 a IS GROUP,                  1 b IS GROUP,
  2 x(i) IS FIELD NUM(4);        2 x(i) IS FIELD NUM(4);
```

The following equation has ambiguity due to the field **x**:

```
y(i) = IF x(i)>10 THEN x(i) - 10
                  ELSE x(i);
```

The ambiguity is resolved by using the qualified (or prefixed) names as follows:

```
y(i) = IF a.x(i)>10 THEN b.x(i) - 10
                    ELSE b.x(i);
```

4.2.2 Incomplete Definitions

If equations or interim data declarations are omitted, the checking system attempts to provide an appropriate equation or a data declaration. The process is based on the following rules [Lu81, SLPP84]:

(1) If an output data node is not explicitly defined, a new equation may be composed using its implicit input nodes.

(2) An omitted data declaration of a node (an interim variable) and/or its parent node can be formulated using its implicit inputs.

If the implicit source of the omitting equations and the declarations are not found in the array graph and/or the dictionary, the system requests the user to provide equations and/or data declarations.

4.2.3 Solvability

A specification without the ambiguous and the incomplete definitions is guaranteed to have at least n equations for n unknown variables. It is one of necessary conditions to have a solution of the equations. Each equation could uniquely determine the value of its LHS variable. If multiple exclusive equations define the same variable, a warning message is issued. A user may respond it by formulating an assertion about the exclusiveness and merging the equations into a single one. For example, consider the following equations:

```
x(i) = IF a(i)>c(i) THEN y(i);
x(i) = IF c(i)<b(i) THEN z(i);
```

To have a unique definition of `x(i)`, the two conditions, `a(i)` $>$ `c(i)` and `c(i)` $<$ `b(i)`, must be exclusive. That is, the following assertion must be true:
$\forall i$, ((((`a(i)` $>$ `c(i)`) & $\neg$ (`c(i)` $<$ `b(i)`)) | ($\neg$ (`a(i)` $>$ `c(i)`) & (`c(i)` $<$ `b(i)`))))
The condition of computing the unique value of `x(i)` becomes an assertion that is regarded as an axiom of the proof system. Since we only allow one equation for one LHS variable, those equations are merged into the following one:

```
x(i) = IF a(i)>c(i) THEN y(i)
                    ELSE IF c(i)<b(i) THEN z(i);
```

The solvability checking is performed by removing the ambiguous and the incomplete definitions of variables and equations as the array graph of a specification is constructed.

4.3 Existence Requirement

As discussed in Section 2.2.6, the existence requirement of variables is the most important property that a MODEL specification must employ. To facilitate a user to comply with the requirement, the checking mechanism provides utilities that examine the *consistent definitions/references* of dimensions and their ranges.

For each equation of the specification, the RHS expression, $f(i_1, ..., i_n, j_1, ..., j_k, var_1, ..., var_k)$ in Figure 2.3, for instance, and its input variables, $var_1, ..., var_k$, have consistent definitions of their dimensions. The LHS variable, $x(i_1, ..., i_n)$, has consistent definitions of dimensions with respect to the equation. The dimension propagation algorithm checks the consistent definitions of dimensions and their references through out the equations of a specification. The ranges of the dimensions are checked by the range propagation algorithms. As a result, the existence condition, namely, $\exists j_1, ..., j_m, var_1, ..., var_k$ and $\exists i_1, ..., i_n$ is checked.

4.3.1 Dimension Propagation

Since some subscripts may be omitted in an equation, it is necessary to check if the dimensionality of arrays referenced in equations is consistent with that of those arrays specified in the respective data declaration. A user does not have to specify the detailed dimensionality of every variable. The checking system completes the data declarations and the equations whose dimensionalities are not explicitly specified.

The checking system propagates attributes of a node of an array graph to another via an edge that connects them. The attributes of an edge stored in the dictionary include the followings [Lu81, Ge89]:

(1) source node of the edge.

(2) its target node.

(3) its type, i.e., hierarchical precedence, data dependency or parameter precedence.

(4) difference of the source and the target nodes in their dimensionality (DIMDIF).

(5) its subscript expression list.

(6) range set for each dimension.

If two nodes are linked by an edge, the attributes of the nodes must be matched according to the attributes of the edge.

An algorithm for the dimension propagation is described in [Lu81]. The dimensionality differences, $DIMDIF$, are set up for all the edges of the array graph. For the input file nodes, the dimensionality is 0. An intermediate node, n, (either a variable or an equation) has an initially declared number of denoting its dimension, D_n. Suppose m source nodes, $s_1, ..., s_m$, are connected to the node, n, via the respective coming edges of dimension differences, $DIMDIF_{s_1}, ..., DIMDIF_{s_m}$. It may have k target nodes, $t_1, ..., t_k$, connected by k outgoing edges of dimension differences, $DIMDIF_{t_1}, ..., DIMDIF_{t_k}$, as shown in Figure 4.1. The current dimensionality of a node, x, is denoted by C_x: the source nodes, $s_1, ..., s_m$, have the dimensionalities,

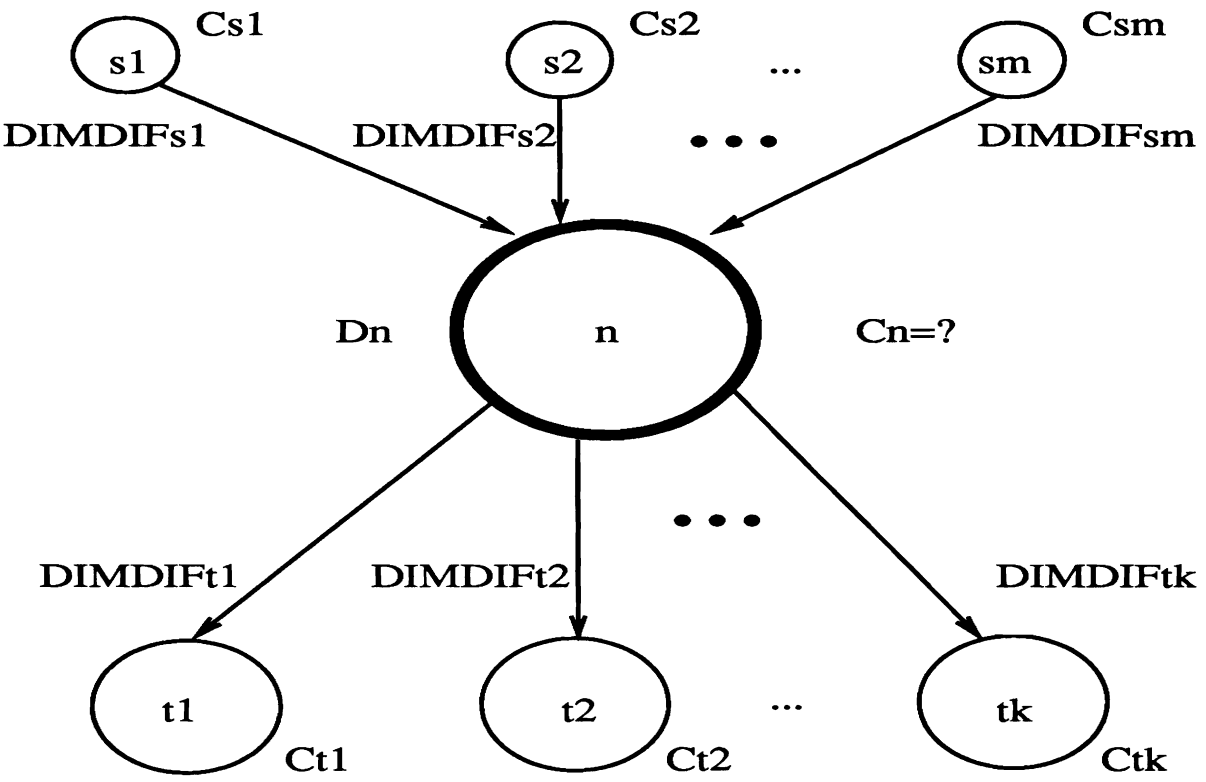


Figure 4.1: Dimension Propagation.

$C_{s_1}, ..., C_{s_m}$ and the target nodes, $t_1, ..., t_k$, have the dimensionalities, $C_{t_1}, ..., C_{t_k}$. Then, the dimensionality of the node n is defined as follows:

$$C_n = max_{1 \leq i \leq m, 1 \leq j \leq k}(D_n, C_{s_i} + DIMDIF_{s_i}, C_{t_j} - DIMDIF_{t_j})$$

The algorithm computes C_n for all nodes of the graph. If every node of the graph has a finite dimension, the algorithm converges [Lu81]. An infinite propagation cycle of the graph can be detected by the algorithm. Then, its nodes and edges are revealed so that a user can fix it. If the dimensionalities of the nodes and the edges are correctly defined, the output file node must have 0 dimensionality.

Missing subscripts of the equations are filled up during the dimension propagation. A node subscript list is formulated for each variable node. Based on these lists, missing subscripts of equation nodes and missing subscript expressions of edges are filled up. The detailed procedure is described in [Lu81].

4.3.2 Range Propagation

After the dimension propagation, the ranges of the dimensions are examined for all the nodes in the array graph. The basic strategy is to find and propagate the user specified ranges of the nodes to the rest of the nodes via the edges connecting them. The propagation aims:

(1) to derive a range for a node subscript not having an explicit range.

(2) to determine *range sets* each of which contains two node subscripts of the same range.

(3) to check the consistent definitions and uses of the ranges.

Definitions

A *node subscript* is defined for a node of an array graph as follows:

(1) $< x, i >$: a node subscript for an i-th (i is a positive integer) dimension of the node of the array variable, x.

(2) $< Eq_n, I >$: a node subscript for I (a subscript variable) with the equation node, Eq_n.

A *range* (or *size*) of a node subscript, $< n, d >$, is defined as $R(< n, d >)$.

User Specified Ranges

A range is specified explicitly or implicitly for each node. It may be explicitly defined by:

(1) a data declaration statement

(2) a subscript declaration statement

(3) the values of control variables ($SIZE$ or END)

(4) the system default: the end-of-file or end-of-record marker ($ENDFILE$) of an input sequential file

Condition of the Propagation

When two node subscripts of different nodes are related through some dependency relation and one of them does not have an explicit range specification, the range of the other node subscript is propagated through the edge denoting the dependency relation.

If a subscript expression $i - k$, where i is a subscript and k is a positive integer, is used in an equation, a *mapping* exists between the values of elements indexed by i and $i - k$. It is assumed that the node indexed by i and the equation node indexed by $i - k$ are in the same range set.

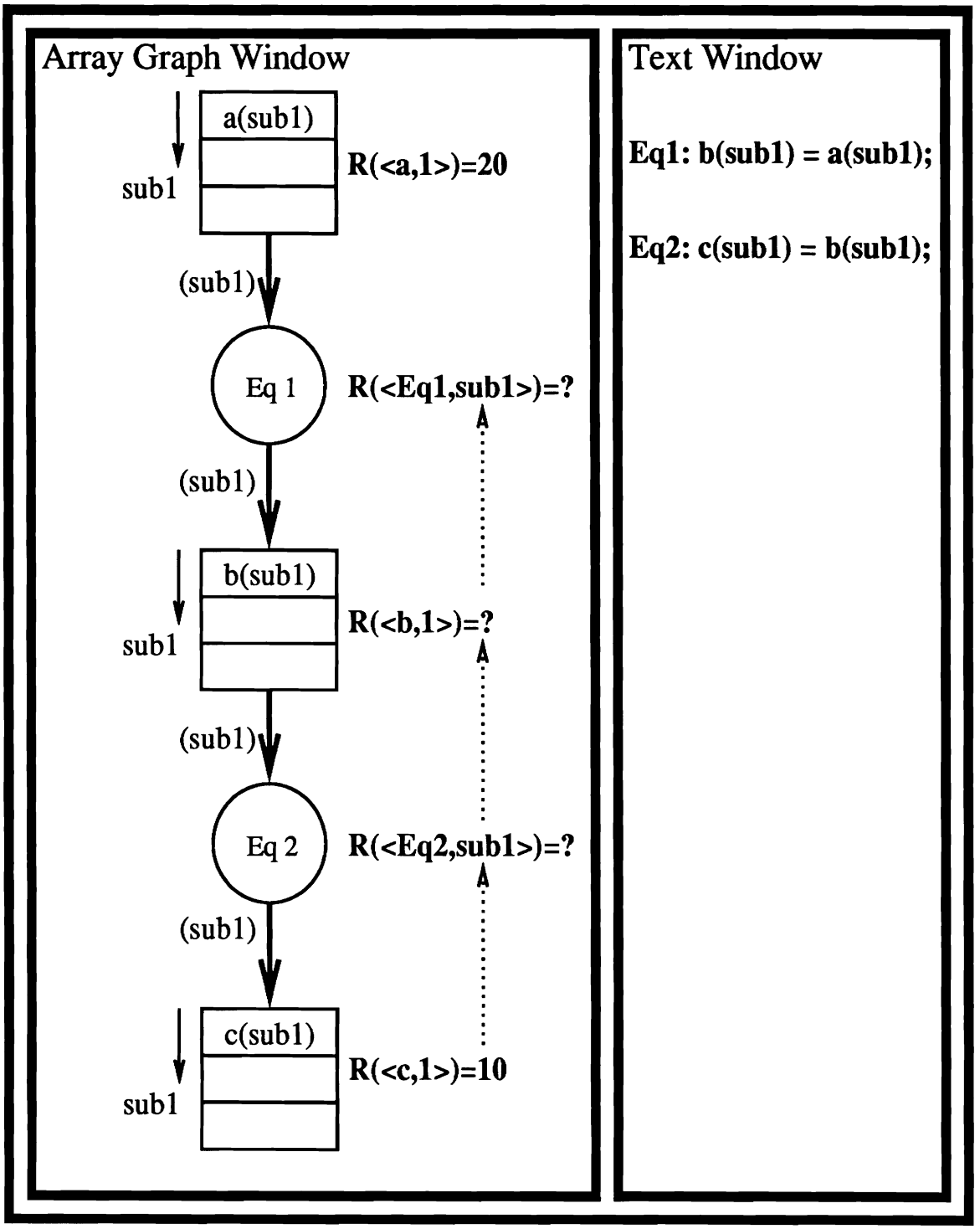


Figure 4.2: Example of Range Propagation.

Priority of the Propagation

There may be many alternatives of range propagation. It is performed based on the following rules:

(1) All the node subscripts with the same *global subscript*[3] are considered as a single group, i.e., a set of variables and statements in a single loop in a procedural language program. Thus the range of the global subscript is propagated with the top priority.

(2) A data array node and its associated control variable such as END and $SIZE$ must have the same range. The range of the control variable is required to be explicitly specified. The range is propagated from the control variable to its data array node with the second priority.

(3) The range of an output node is propagated to its associated equation node with the second priority.

(4) From an equation node to its associated input data node, the range can be propagated. It has the third priority.

(5) The lowest priority is given to the range propagation from an input data node to its equation node.

Consider a simple example illustrated in Figure 4.2. Two simple equations, `Eq 1` and `Eq 2`, of transferring values from an array to another are presented. The ranges of the arrays, `a(sub1)` and `c(sub1)`, are given as 20 and 10, respectively. Note that the subscript, `sub1`, is not defined as a global subscript. Since the condition of the range propagation is satisfied, we can propagate the ranges to determine the ranges of the node subscripts for `Eq 1`, `Eq 2` and `b(sub1)`. We have the following alternatives: (a) propagate the range of the local subscript `sub1` of `a(sub1)` to the equation node, `Eq 1`, to determine the value of `R(<Eq 1,sub1>)` or (b) propagate the range of the subscript `sub1` of the output node, `c(sub1)`, backward to the equation node, `Eq 2`, to get the value of `R(<Eq 2,sub1>)`. According to the rules of the range propagation, the first alternative, (a), has the fourth priority and the second alternative, (b), has the second priority. Thus the value of `R(<Eq 2,sub1>)` is defined as 10. Next, we have the following two alternatives: (a) and (c) propagate the value of `R(<Eq 2,sub1>)` to its input data node, `b(sub1)`. The alternative, (c), has higher priority. Therefore the value of `R(<b,1>)` becomes 10. Finally, the following two alternatives remain: (a) and (d) propagate the range of the subscript, `sub1`, for the output node, `b(sub1)`, to its equation node, `R(<Eq 1,sub1>)`. The second alternative, (d), has the second priority. It follows that `R(<Eq 1,sub1>)` is equal to 10.

[3]defined by either a subscript declaration statement or a control variable, *FOR_EACH* [Lu81, MOD89].

Range Functions and Real Arguments

A node subscript represents an iteration over its range by a loop control statement in a procedural program [Lu81]. An equation and a data node in an array graph may have multiple node subscripts and they represent a multi-level nested loop. In such a situation, the range of a node subscript can be a function of the other subscripts. For example, consider the following MODEL specification:

```
a IS FIELD;
b IS FIELD;

Eq 1: b(i,j,k) = a(i,j,k);
Eq 2: SIZE.a(i,j) = f(i,j);
```

The range of the third dimension, `k`, of the array variables, `a(i,j,k)` and `b(i,j,k)`, depends on the ranges of the first and the second dimensions, `i` and `j`, as `Eq 2` defines. The specification is translated into the following code of a procedural language program [Lu81]:

```
DO <a,1>;
   DO <a,2>;
      DO <a,3> = 1 TO SIZE.a(<a,1>,<a,2>);
         b(<a,1>,<a,2>,<a,3>) = a(<a,1>,<a,2>,<a,3>);
      END;
   END;
END;
```

An n-dimensional *range array*, $SIZE.x(i_1,...,i_n)$, is regarded as a *range function*. The range function accepts integer arguments, $i_1,...,i_n$, and computes the range of the `n+1`-th or higher dimension of the variable, x. Arguments of a range function are called *real arguments*, if they really contribute to determining the value of the function. An algorithm of finding real arguments of range functions is described in [Lu81].

It is required that the loops of an array are nested according to the sequence of the array dimensions. That is, the loops of a variable, $x(i_1,...,i_n)$ must be nested in the following way:

```
DO <x,1>;
   DO <x,2>;
      ...
         DO <x,n>;
            ...
         END;
      ...
```

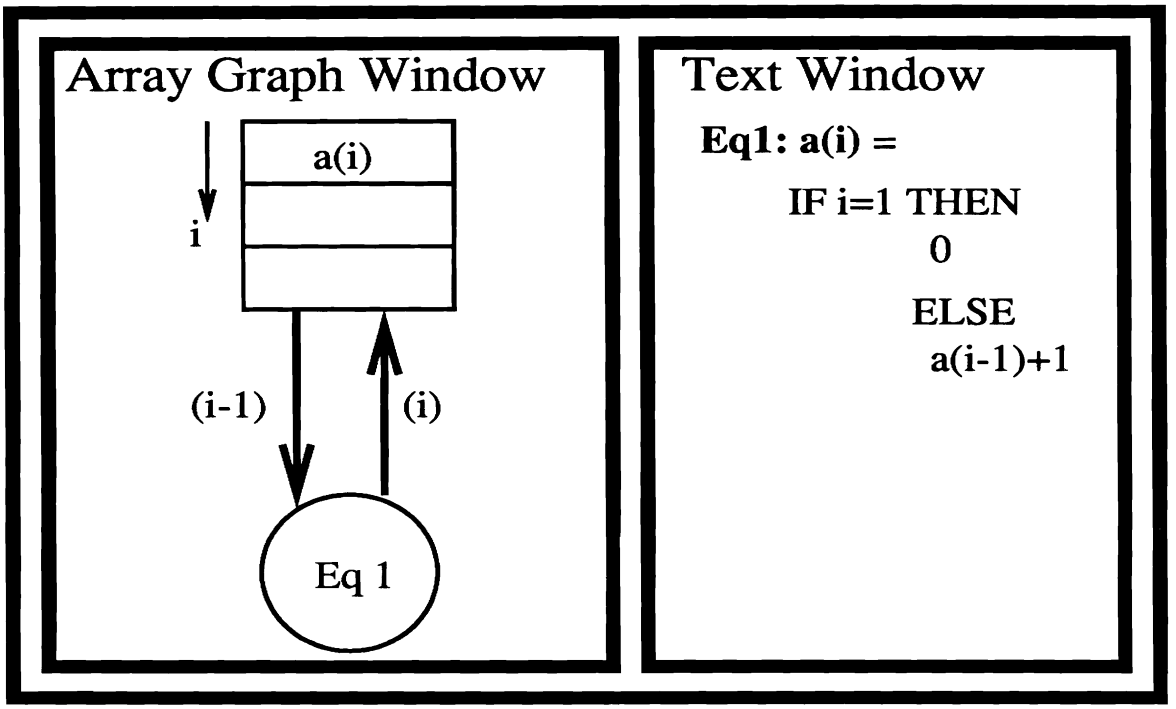


Figure 4.3: A maximally strongly connected component (MSCC) of an array graph.

```
   END;
END;
```

It follows that a range function for a dimension, i_k, $1 \leq k < n$, does not affected by its lower dimensions for all i_m, $k < m \leq n$.

Range Propagation Algorithm

There are three basic algorithms for the range propagation. The first algorithm locates the user specified ranges of node subscripts. As discussed before, the ranges are specified by declaration statements (either data or subscript), the control variables such as *END* and *SIZE* or the system default (end-of-file or end-of-record). Secondly, the explicit range specifications are propagated. It requires the node subscripts to be partitioned into their corresponding range sets. Finally, the real argument list is formulated for the node subscripts in the same range set and is propagated. See [Lu81] for the details of the algorithms.

4.4 Causal Chain

A causal chain in the array graph of a specification is a path from its input nodes to the output nodes via the equations. A solution of the equations is computed along

the causal chain. Therefore, a circular definition of variables, namely *circular logic* (or *dependency*), that cause an infinite computation, should not be on a causal chain.

A maximally strongly connected component (MSCC) of an array graph could result in circular logic. However, not all MSCC's form circular logic. The checking mechanism identifies and decomposes an MSCC by deleting edges that represent data dependencies assured by iteration statements [Lu81, SLPP84]. Such edges of iteration can be determined by examining the subscript expression of the edges. If its subscript expression is in the form of $sub - k$, where $k > 0$ and sub denotes a subscript common to all MSCC nodes, the edge is determined to be representing iteration.

For example, an MSCC can be found in Figure 4.3: a cycle formed by the edge labeled with `i` and the edge with `i-1`. In this particular example, $sub = i$ and $k = 1$. Therefore, the edge with `i-1` is classified as one representing iteration. It follows that the MSCC does not have circular logic. Such an *iteration solution* method is recursively applied until all MSCC's in the array graph are examined.

A cycle that cannot be decomposed by the iteration solution method is reported as a possibly infinite loop. A user has to examine and remove such a cycle by decomposing it. If it is not possible, he may use a set of simultaneous equations that perform the same function of the cycle. In general, it is very complicated to remove an infinite loop from a program by static checking. We only deal with a specification that always has at least one causal chain of the equations.

4.5 Termination

Suppose we have an acyclic array graph without any cycle. Then a causal chain can be formulated. However, it does not mean that a solution of equations is obtained. To check the termination condition, the equations defining control variables such as END and $SIZE$ are examined for each causal chain. It aims to check the *finiteness* of the subscripts, namely, $1 \leq i_1 \leq SIZE_{i_1}, ..., 1 \leq i_n \leq SIZE_{i_n}$ and $1 \leq j_1 \leq SIZE_{j_1}, ..., 1 \leq j_m \leq SIZE_{j_m}$ of the existence condition in Section 2.2.6. The values of the $SIZE$ variables (= the ranges of the subscripts), if computable, may be obtained during the range propagation. There may exist some constraints of defining the ranges that cannot be computed during the range propagation. Such constraints are discovered and checked during the termination checking.

Note that the END variable can define a minimum range of 1 because it must have at least one boolean value. However, the $SIZE$ variable can have a minimum range of 0. The value of the END variable can be infinite while the $SIZE$ variable has a finite value. $END.x(i_1, ..., i_n)$ may depend on the values of the array, $x(i_1, ..., i_n)$. But $SIZE.x(i_1, ..., i_n)$ must be computed before any element of $x(i_1, ..., i_n)$ is used.

The termination checking discovers an equation of defining the size of the array variables such as "$END.x(i_1, ..., i_n) = ...$" and "$SIZE.x(i_1, ..., i_n) = ...$" for each causal

chain of a specification. If there is no equation of such control variables, a warning message is issued so that a user examines the termination condition of a specification. Though such an equation is defined on a causal chain, it may not have an explicit finite value denoting the termination. In such a case, a warning message is also issued.

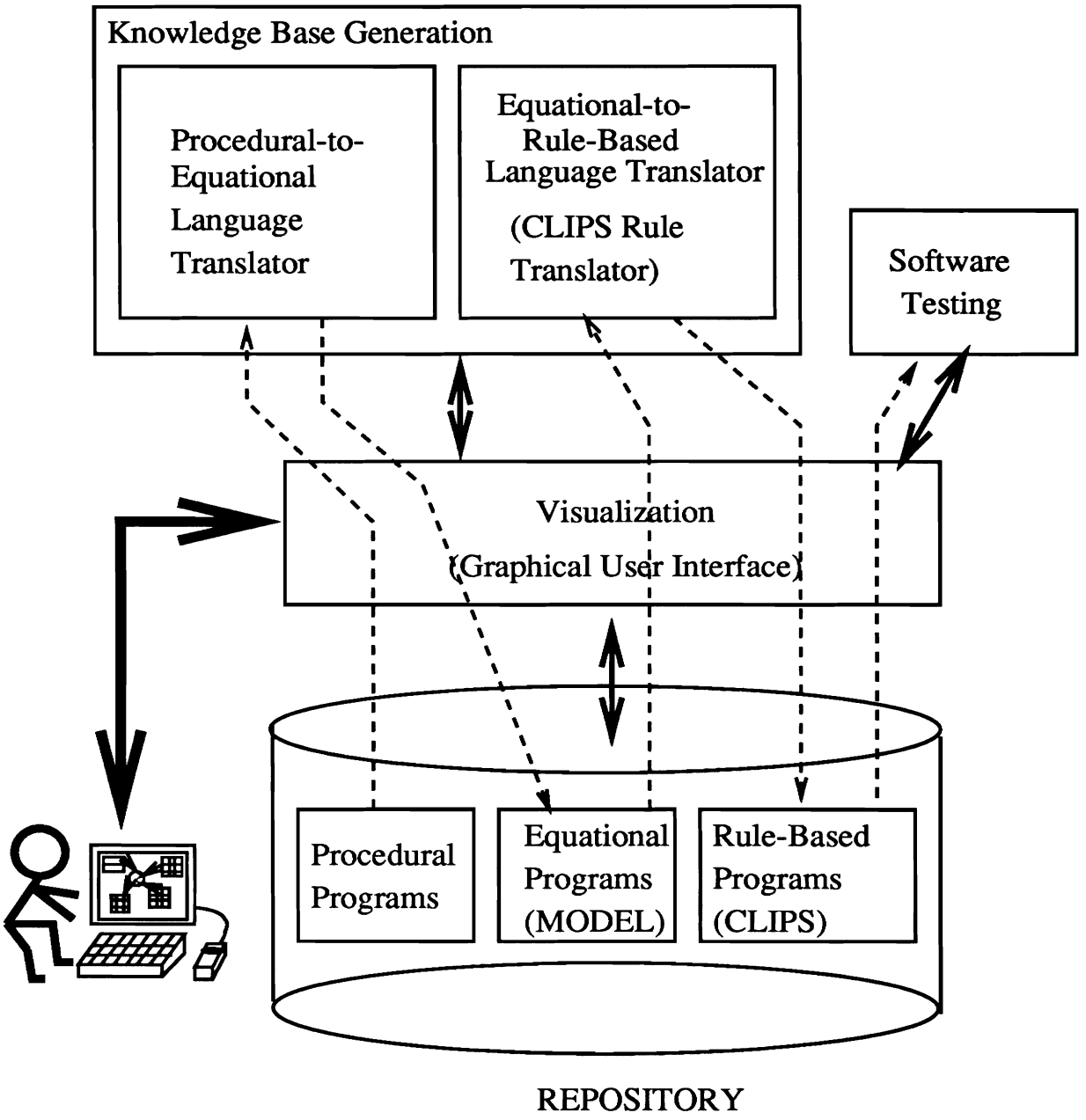


Figure 5.1: Extracting expertise from programs, converting it to rules, and accumulating the rules into knowledge base.

and a graphical user interface. Each MODEL equation is regarded a functional unit expressed as an equation node of an array graph that is used as a graphical user interface during the testing. A MODEL variable is denoted by a data node of the array graph. The data flows between the equation nodes and the data nodes are expressed by edges of the array graph. A human tester can examine inputs and outputs of an individual equation (called *I/O testing*) and/or a number of equations along the selected data flows (called *path analysis*) specified in the array graph.

This section presents introductory descriptions of the proposed testing method in use of an expert system: The objectives of using an expert system for software testing are discussed. The advantages of the testing based on equational language programs (MODEL specifications) over procedural language programs are also presented. The procedure starts with the translation of a MODEL equation into a CLIPS rule using the CLIPS translator. This is also used for the knowledge base generation. Section 5.2 explains the translator. The translation process is illustrated using the specification of the gcd example[1] into CLIPS rules in Section 5.3. The interactive procedures of the I/O testing and the path analysis are illustrated in Section 5.4.

5.1.1 Objective of the Use of an Expert System

We build a processor that directly executes MODEL equations so that the testing of the equations (a MODEL specification) is possible using such a processor. A data flow machine could be the processor, since it can be envisaged as an execution model of MODEL, as discussed in Chapter 2. Also an expert system could serve as a MODEL processor because an expert system rule can simulate an execution of a MODEL equation. Recall Section 2.2.7 for details. We use an existing expert system called CLIPS [GR89] as a MODEL processor. With the MODEL processor, we can perform the software testing at a specification level.

MODEL equations of a specification can be translated into expert system rules as will be shown in Section 5.1.2. The expert system executing the translated rules is able to compute the same outputs for the same input data as the MODEL equations do. Therefore the expert system can simulate functional behaviors of the MODEL specification as it executes the translated rules. It follows that a human tester can do the software testing of the specification using the expert system. As illustrated in Figure 5.2, a MODEL specification is translated into a set of expert system (CLIPS) rules by the CLIPS rule translator. Then the expert system fires the rules according to the requests from the user. He designates the equation(s) which he wants to evaluate and enters the values of the associated input variables. Then the system computes the output(s) of the tested equation(s) by executing the corresponding rule(s) translated from the equation(s). It is usually assumed that he know the desired output values from the given input data set during the software testing [RW85, How87]. The interaction between the user and the system is performed through the array graph of the

[1] presented in Section 2.4 of Chapter 2

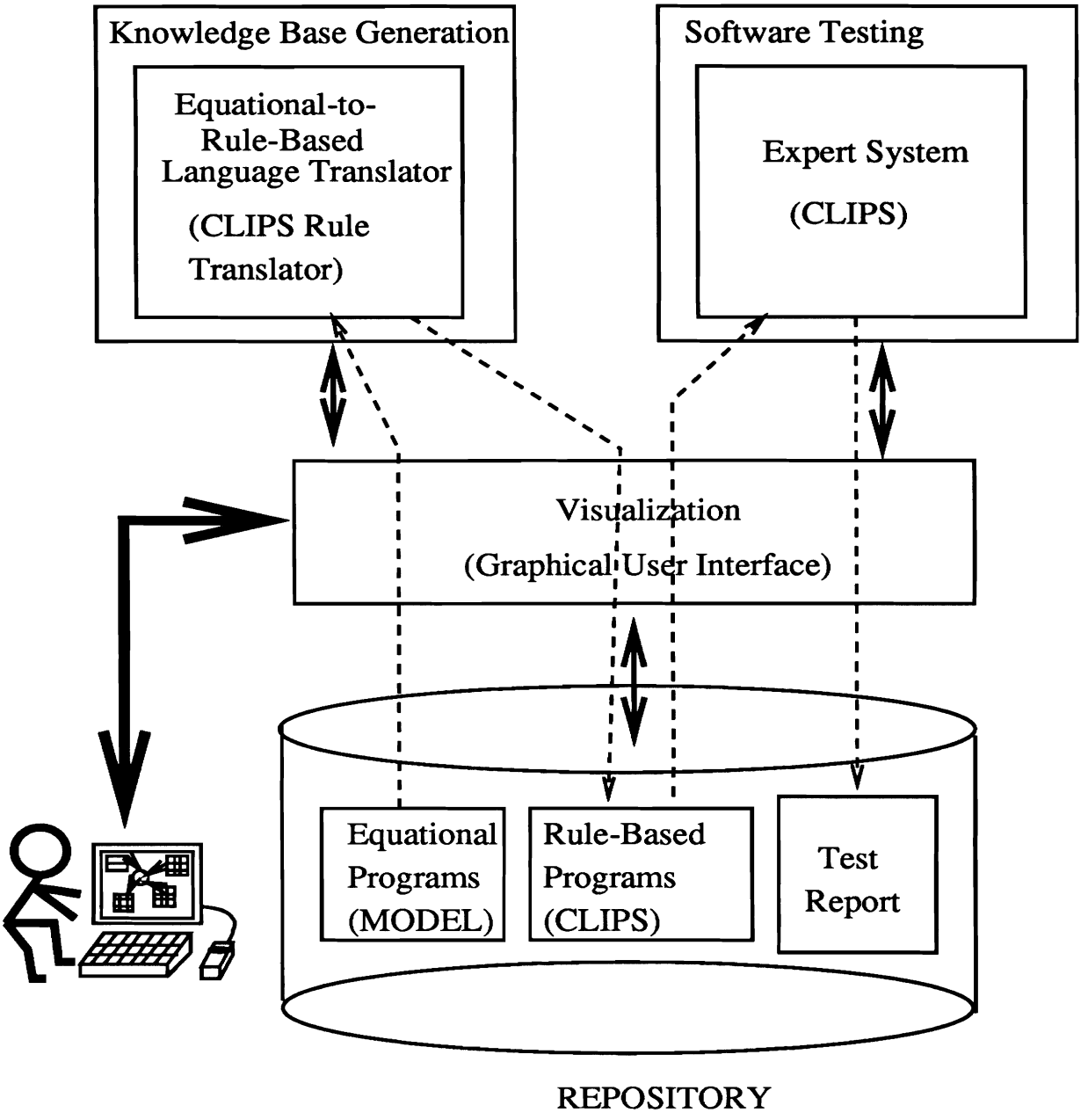


Figure 5.2: Software testing at a specification level using an expert system.

specification via the graphical user interface. The user types the test input values using a keyboard and a mouse. The values of the computed output variables are displayed through the graphical user interface. CLIPS is also able to "explain" how to compute the outputs according to the test inputs. The textual and the graphical results of the testing are stored as test reports in the data base (the repository of the software developing environment).

As discussed in Chapter 3, an array graph of a MODEL specification is visualized and serves as a graphical user interface of the software developing environment. It is also used as a graphical user interface for the testing. We claim that the interaction between the user and the environment is enhanced. Also, it is possible to produce both the textual and the graphical forms of the test reports. The textual form includes test input data sets and their generated output values. The coverage of each test data set for either a part of or a whole specification is marked on the displayed array graph.

5.1.2 Similarity between a MODEL equation and a CLIPS rule

How an expert system rule can simulate a MODEL equation? As an example, consider the following equation, `Eq 1` in Figure 2.4, Chapter 2:

```
Eq 1: y1(i) = IF i=1 THEN IF x1 > x2 THEN x1 ELSE x2
                     ELSE IF y1(i-1) > y2(i-1) THEN y1(i-1) - y2(i-1)
                                               ELSE y1(i-1);
```

The equation accepts the input values of variables, $x1, x2, y1(i)$ and $y2(i)$, where $1 \leq i \leq SIZE.y1$ and computes the values of $y1(i)$ for $1 \leq i \leq SIZE.y1$. As discussed in Chapter 2, the existence condition of `Eq 1` must be:

$$\forall i, 1 \leq i \leq SIZE.y1, (\exists x1, x2 | \exists y1(i-1), y2(i-1))$$

It means that the values of $x1$ and $x2$ OR $y1(i-1)$ and $y2(i-1)$ must be available when the value of $y1(i)$ is determined by the equation, `Eq 1`. The existence condition can be satisfied either $\exists x1, x2$ or $\exists y1(i-1), y2(i-1)$ not both. It is because the first elements of the array variables, $y1(1)$ and $y2(1)$, are initialized by $x1$ and $x2$, respectively, and the rest elements of the arrays, $y1(i), y2(i)$, for $2 \leq i \leq SIZE.y1$, are computed from their "ancestor" elements, $y1(i-1)$ and $y2(i-1)$. *Scheduler* of the MODEL compiler is able to detect such data dependency by examining array graphs [Lu81, MOD89]. It can statically decompose such a complex equation into the following simple equations:

```
Eq 1-1: y1(1) = IF x1 > x2 THEN x1 ELSE x2;
Eq 1-2: y1(i) = IF y1(i-1) > y2(i-1) THEN y1(i-1) - y2(i-1)
                                     ELSE y1(i-1);
```

`Eq 1-1` is similar to an initialization statement of an iteration block while `Eq 1-2` can be regarded as a body of the iteration block. Then the existence conditions for `Eq 1-1` is $\exists x1, x2$. On the other hand, the equation, `Eq 1-2`, has the existence condition such that $\forall i, 2 \leq i \leq SIZE.y1, \exists y1(i-1), y2(i-1)$.

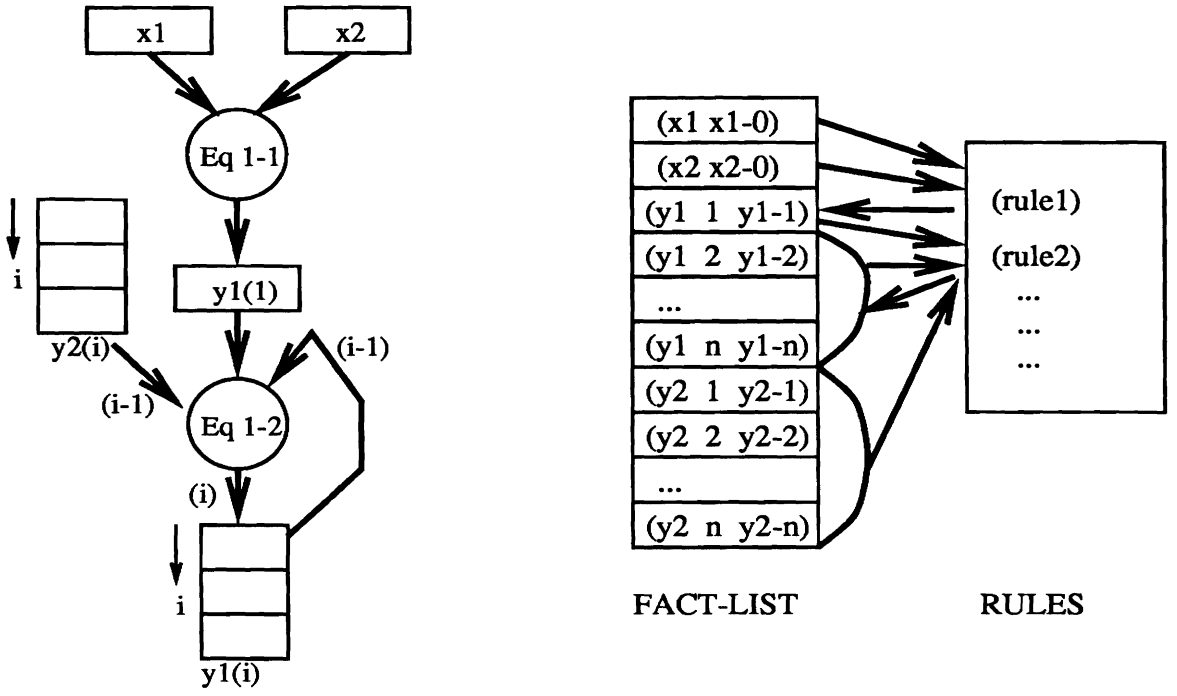
We translate a MODEL equation into a CLIPS rule that performs the same function as the equation does. A CLIPS rule, in general, has two components: preconditions and actions. Whenever the preconditions of the rule are satisfied, the actions are executed. The CLIPS expert system maintains a list of facts stored in its knowledge base, called *fact-list* in CLIPS. The satisfiability of the preconditions is tested through *pattern matching* [GR89]. We propose that the existence conditions of a MODEL equation are translated into preconditions of a CLIPS rule; The body of the equation is denoted by the actions of a CLIPS rule; Every element of array variables in MODEL is represented by a CLIPS fact in the fact-list; A MODEL subscript is expressed as a variable in CLIPS.

A set of MODEL equations are translated into a collection of CLIPS rules. Input values of the equations are provided as facts for CLIPS. Then, the expert system performs the pattern matching with the patterns specified in the precondition part and the facts (the input values for the equations) stored in the fact-list. It checks if the facts of such patterns are in the fact-list (if the required inputs are available). If so, the values of the variables specified in the precondition part are obtained from the facts. It follows that the rules, whose preconditions are satisfied, are invoked. A set of new facts (the outputs of the equations) are generated and the fact-list is updated.

A CLIPS rule, `rule1`, which will be shown to be equivalent to the equation, `Eq 1-1`, is defined as follows:

```
(defrule rule1       ; for Eq 1-1
;  ''preconditions''
   (x1 ?x1)          ; get initial values of x1 and x2
   (x2 ?x2)
   =>
;  ''actions''
   (if (> ?x1 ?x2) then
             (assert (y1 1 ?x1))
                   else
             (assert (y1 1 ?x2))))
```

The variables, $x1$ and $x2$, are represented as CLIPS facts, `(x1 x1-0)` and `(x2 x2-0)` in Figure 5.3-(b), where `x1` and `x2` are relation names of the facts and `x1-0` and `x2-0` are their values. The CLIPS variables denote the values of the input variables of the equation, $x1$ and $x2$. The existence conditions of the equation, $\exists x1, x2$, are encoded as the preconditions of the rule. The body of the equation is translated into the actions of the rule. As shown in Figure 5.3, `rule1` defines a new fact, `(y1 1 y1-1)`, where `y1-1` is a value computed by the actions of the rule, from the input facts, `(x1 x1-0)` and



(a) execution of MODEL equations (b) execution (pattern matching) of CLIPS rules

Figure 5.3: MODEL equations and their translation into CLIPS rules.

`(x2 x2-0)`. It is equivalent to the computation such that the equation, `Eq 1-1` defines the value of $y1(1)$ from $x1$ and $x2$. For example, the equation computes $y1(1) = 65$ if $x1 = 26$ and $x2 = 65$. The execution can be simulated by CLIPS as follows: The input values are provided by commands, `(assert (x1 26))` and `(assert (x1 65))`. The satisfiability of the preconditions is checked through the pattern matching and the CLIPS variables, `?x1` and `?x2`, get the values, 26 and 65, respectively. Since the preconditions are satisfied, the actions are executed. In this particular example, CLIPS command, `(assert (y1 1 ?x2))`, is invoked. Since `x2` is asserted as 65, a new fact, `(y1 1 65)`, is formulated and stored in the fact-list, as shown in Figure 5.3-(b).

`Eq 1-2` is translated into the following rule, `rule2`:

```
(defrule rule2    ; for Eq 1-2
   (end-y1 =(- ?i 1) ?val&:(not ?val)) ; check if END.y1(i-1) is false
                          ; ?val denotes the truth value of END.y1(i-1)
                          ; if ?val is TRUE, rule2 is not executed.
   (y1 =(- ?i 1)&(> ?i 1) ?y1)   ; y1 existence condition, namely,
                                 ; the value of y1(i-1) exists and
                                 ; the subscript i is greater than 1.
   (y2 =(- ?i 1)&(> ?i 1) ?y2)   ; y2 existence condition, namely,
                                 ; the value of y2(i-1) exists and
                                 ; the subscript i is greater than 1.
   =>
```

```
(if (> ?y1 ?y2) then
                (assert (y1 ?i =(- ?y1 ?y2)))
                else
                (assert (y1 ?i ?y1))))
```

The array variables, $y1(i)$ and $y2(i)$, of the equation are represented by multiple argument facts like `(y1 i y1-i)` and `(y2 i y2-i)`, $1 \leq i \leq n$, as shown in Figure 5.3-(b). `y1` and `y2` denote relation names of the facts. The first argument, `i`, represents the subscript of the array variables (the index of the facts). The second arguments, `y1-i` and `y2-i`, denote the values of the elements of the array variables, $y1(i)$ and $y2(i)$, respectively. The existence conditions of `Eq 1-2` contains $\forall i, 2 \leq i \leq SIZE.y1$. Note that we determine the value of $SIZE.y1$ using the control variable, $END.y1$, such that $\forall i, 1 \leq i < SIZE.y1, END.y1(i) = false$ and $END.y1(SIZE.y1) = true$. Precondition, `(end-y1 =(- ?i 1) ?val&:(not ?val))`, checks the existence condition. It tests if there is a fact named `end-y1`, if the first argument of the fact is equivalent to the value of `=(- ?i 1)`, and if its second argument, `?val`, such that `(not ?val)` is *true*, exists. When `?val` is *false*, the pattern matching succeeds because it makes `(not ?val)` *true*. That is, it checks if $END.y1(i-1) = false$. `(y1 =(- ?i 1)&(> ?i 1) ?y1)` means that there must exist $y1(i-1)$ such that i is greater than 1. If so, the expert system assigns a value to the variable, `?y1`. Similarly, `(y2 =(- ?i 1)&(> ?i 1) ?y2)` checks the existence condition of $\exists y2(i-1)$. The actions of the rule represent the consequence of executing the equation. New facts, `(y1 2 ?y1)`, ..., `(y1 n ?y1)`, where `n` $= SIZE.y1$, are generated by the rule and they represent the elements of the array variable, $y1(2), ..., y1(n)$.

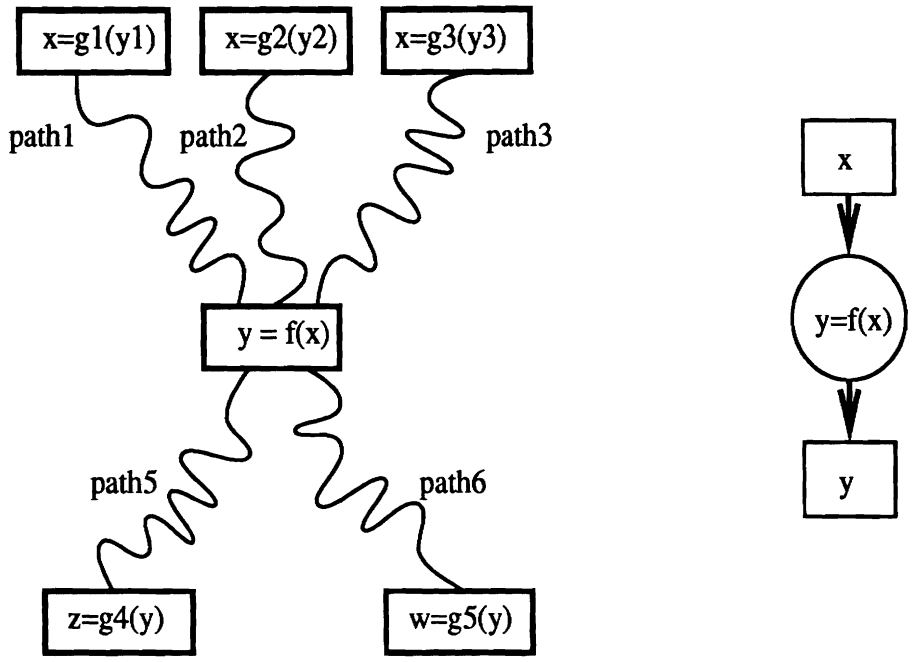
We can find the following similarities of the two systems:

(1) The variables of the equation can be represented by the facts of the expert system: The value of a MODEL variable cannot be changed after it is "assigned" (using "=") by an equation. Similarly, the fact of the expert system cannot be modified either once it is "asserted" (using the `assert` command) by a rule.

(2) The existence condition of the equation can be regarded as the preconditions of the rule: The equation is fired only when all of its inputs are available (the existence condition is true). Similarly, the rule is executed only when all of the preconditions, which check the existence of facts denoting the MODEL input variables, are satisfied.

(3) The consequences of executing the equation can be simulated by the actions of the rule: The assertion of new facts in the expert system is equivalent to the assignment of values to the outputs in MODEL.

It concludes that the execution of a MODEL equation is similar to the execution of a rule in the expert system, CLIPS. Therefore, an array graph for a MODEL specification can be translated into the sequence of pattern matching scheduled by the Rete algorithm [For82, GR89]. In many rule-based expert systems, the Rete algorithm is



(a) Data flow testing in procedural programs. (b) Testing of an equation node and its data nodes

Figure 5.4: Data Flow Analysis

used in scheduling efficient pattern matching in a large collection of rules and facts. Normally, the fact-list of expert systems are modified during each cycle of the pattern matching process [GR89]. And the changes of the fact-list during each cycle are typically small percentage of the whole fact-list. Therefore we can reduce unnecessary computations of searching for facts by having the new or updated facts search for rules instead of having rules search for the whole facts in the fact-list. The details of the algorithm is described in [For82, GR89].

5.1.3 Procedural versus Equational Testing

Traditional software testing methods of procedural program codes are based on *program-path analysis* [DLS78, DMMP87, How87, Ham88, Bei90]. It checks the changes of the values of program variables (= states) along the control flows of program (= program-paths). The basic idea is to execute paths of a program as a part of testing it.

One such approach is called *data flow testing*. It aims to test the consequences of each and every computation using the produced values [RW85, Bei90]. As illustrated in Figure 5.4-(a), the data flow testing focuses on definition of variables and their uses. For example, the value of the variable, `x`, could be determined by the multiple statements, `x=g1(y1)`, `x=g2(y2)` and `x=g3(y3)`, as shown in the definition/use graph [RW85] of Figure 5.4. We must invent a test data set that covers all paths, `path1`, `path2` and `path3`, between the definitions of the values for `x` and their use, `f(x)`, in the data flow

testing [RW85]. Similarly, variable, `y`, that is defined by the statement, `y=f(x)`, can be used by multiple statements, `z=g4(y)` and `w=g5(y)` as shown in Figure 5.4-(a). The paths must be covered by the test data. In contrast, there is only one equation that determines the value of the variable, `x`, in an equational language program. Figure 5.4-(b) illustrates the characteristic of equational languages such as MODEL. Since a single equation uniquely defines the value of a MODEL variable and a MODEL equation is executed only when all of its inputs are available, we only check the paths between the equation, `y = f(x)`, its input variable nodes, `x`, and its output node, `y`.

The analysis on an equational language program is less complicated than that on a procedural language program: The number of paths to be analyzed must be smaller. When the testing is performed via a graphical representation and the results are reported in graphics, the equational testing offers simpler diagrams than the procedural testing does.

5.2 Translation

A MODEL variable is represented by a CLIPS fact: A subscript and a scalar are expressed by single argument facts. An array is denoted by a multiple argument fact. A MODEL equation is translated into CLIPS rule(s) depending on its existence conditions. The procedure of the translation is described in this section. An example of translation will be given in the next section.

The translation is performed in the following steps:

(1) Data declaration is translated: A subscript and a scalar variable are translated into single argument facts. A multi-dimensional array is converted to a multiple argument fact.

(2) The existence conditions of each equation are examined. Depending on the structure of the existence conditions, an equation may have to be decomposed. The scheduler of the MODEL compiler can statically decompose such a complex equation into simple ones each of which can be represented by a single CLIPS rule.

(3) The existence conditions of each equation are translated into the preconditions of the corresponding rule.

(4) The equation body of each equation is translated into the actions of the corresponding rule.

A MODEL variable is stored as a CLIPS fact. A scalar variable, `x1`, is represented by a CLIPS fact. Thus, a MODEL equation such as `x1 = 26;` is translated into an assertion of a new fact, `(assert (x1 26))`. On the other hand, the value can retrieved by the following command, `(x1 ?x1)`, where `x1` is the relation name and `?x1` is a variable. Through the pattern matching between the asserted fact, `(x1 26)`

and the command, `(x1 ?x1)`, the variable has the value, `26`. A subscript variable is also denoted by a single argument fact. An element of an n-dimensional array such as $\mathtt{w}(\mathtt{i}_1, \mathtt{i}_2, ..., \mathtt{i}_n)$, where $\mathtt{i}_1, \mathtt{i}_2, ..., \mathtt{i}_n$ are subscripts, is expressed as a CLIPS fact such as (`w` i_1 i_2 ... i_n `val`), where `val` is the value of the element. Its definition and retrieval are as same as those of the scalar variable.

A MODEL function is translated into a CLIPS function using the `deffunction` command.

A MODEL equation consists of two parts: the implicit existence condition and the equation body. The existence condition is translated into the preconditions of a CLIPS rule. The equation body becomes the actions of the rule.

5.3 Example

The equations of the gcd example shown in Figure 2.4, Chapter 2, are translated into CLIPS rules to illustrate the translation procedure.

As illustrated in Section 5.1.2, equation, `Eq 1`, has to be decomposed into two simple ones, `Eq 1-1` and `Eq 1-2`, due to its complex existence condition. The simple equations are translated into CLIPS rules, `rule1` and `rule2`, respectively. Similarly, equation, `Eq 2`:

```
Eq 2: y2(i) = IF i=1 THEN IF x1 > x2 THEN x2 ELSE x1
                     ELSE IF y1(i-1) > y2(i-1) THEN y2(i-1)
                                               ELSE y2(i-1) - y1(i-1);
```

should also be decomposed into the following two simple equations:

```
Eq 2-1: y2(1) = IF x1 > x2 THEN x2 ELSE x1;
Eq 2-2: y2(i) = IF y1(i-1) > y2(i-1) THEN y2(i-1)
                                     ELSE y2(i-1) - y1(i-1);
```

The existence conditions for `Eq 2-1` is $\exists x1, x2$. The equation is translated into the following rule:

```
(defrule rule3       ; for Eq 2-1
   (x1 ?x1)          ; get initial values of x1 and x2
   (x2 ?x2)
   =>
   (if (> ?x1 ?x2) then
            (assert (y2 1 ?x2))
                  else
            (assert (y2 1 ?x1))))
```

The equation, Eq 2-2, has the following existence condition:

$$\forall i, 2 \leq i \leq SIZE.y2, \exists y1(i-1), y2(i-1)$$

The rules are translated into the following rules:

```
(defrule rule4      ; for Eq 2-2
   (end-y1 =(- ?i 1) ?val&:(not ?val)) ; check if END.y1(i-1) is false
                          ; ?val denotes the truth value of END.y1(i-1)
                          ; if ?val is TRUE, rule2 is not executed.
   (y1 =(- ?i 1)&(> ?i 1) ?y1)    ; y1 existence condition, namely,
                                  ; the value of y1(i-1) exists and
                                  ; the subscript i is greater than 1.
   (y2 =(- ?i 1)&(> ?i 1) ?y2)    ; y2 existence condition, namely,
                                  ; the value of y2(i-1) exists and
                                  ; the subscript i is greater than 1.
   =>
   (if (> ?y1 ?y2) then
                  (assert (y2 ?i ?y2))
                  else
                  (assert (y2 ?i =(- ?y2 ?y1)))))
```

Those four rules, rule1 to rule4, compute the facts, $\forall i, 1 \leq i \leq SIZE.y1$, (y1 i y1-i), (y2 i y2-i), where y1-i and y2-i represent the values of y1(i) and y2(i), respectively.

The equation, Eq 3:

```
Eq 3: END.y1(i) = (y1(i) = y2(i));
```

becomes the following rule:

```
(defrule rule5
   (y1 ?i ?y1) ; y1 existence condition
   (y2 ?i ?y2) ; y2 existence condition
   =>
   (assert (end-y1 ?i =(= ?y1 ?y2))))
```

The multiple argument fact, end-y1), contains boolean values in its second argument. The boolean value is determined by evaluating expression, =(= ?y1 ?y2). The expression returns *true* if the values of ?y1 and ?y2, which represent the MODEL variables, $y1(i)$ and $y2(i)$, respectively, are same. Otherwise, it returns *false*. The first argument, ?i, is an index of the fact which is in fact a subscript of a MODEL variable, $END.y1$. Since $y1(i) \neq y2(i)$ for all $i, 1 \leq i \leq SIZE.y1$, the fact looks

like as follows: `(end-y1 1 false), (end-y1 2 false),..., (end-y1 SIZE.y1 - 1 false), (end-y1 SIZE.y1 true).`

Finally, the equation, `Eq 4`:

```
Eq 4: z = IF (END.y1(i)) THEN y1(i);
```

becomes the following rule:

```
(defrule rule6
   (y1 ?i ?y1)          ; y1 existence condition
   (end-y1 ?i ?end-y1) ; end-y1 existence condition
   =>
   (if (?end-y1) then
        (assert (z ?y1))))
```

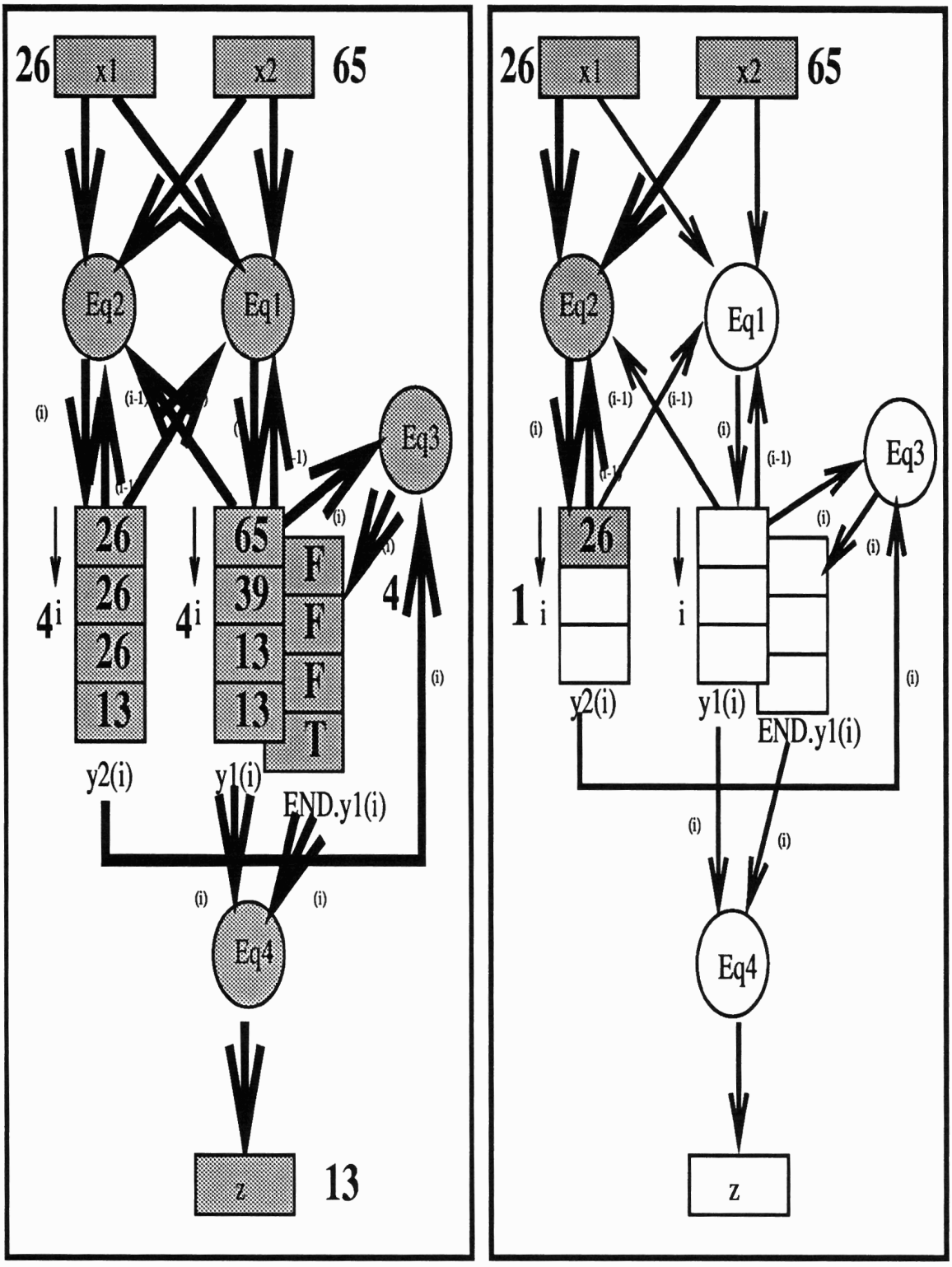
5.4 Interactive I/O Testing and Path Analysis

The basic notion of the software testing is to execute a MODEL specification with a test input data set and analyze the results. The default method of the testing is such a testing of the whole specification. A human tester, however, may want to test individual equation (I/O testing) or a set of equations along data flows (path analysis). In that case, he may set the testing mode as either "I/O TESTING" or "PATH ANALYSIS" using a pull-down menu. During the I/O testing of an individual equation, he designates the equation being tested and supply inputs for it. For the path analysis, he chooses a set of equations on some data paths to be tested. Then a selected test input data set is provided for the analysis.

It is required that the expert system, CLIPS, activates all rules and facts for the specification before the testing. A human tester must choose a test input data set that can cover all or the part of the MODEL specification being tested. It is assumed that the desired output values according to the inputs are known. Thus the teser can decide whether or not the specification works as desired by reviewing the outputs from the given test inputs.

The interaction is performed through the array graph of the specification via the graphical user interface. That is, test input data are provided via the input data nodes of the displayed array graph and the outputs are shown through the output data nodes of the graph.

Suppose a human tester has a test input set, $x1 = 26$ and $x2 = 65$, for the gcd test program of Figure 2.4, Chapter 2. He would clicks the nodes of $x1$ and $x2$ using a mouse and supplies the inputs, 26 and 65, via the displayed array graph. They are automatically converted by the system as CLIPS assertions such as `(assert (x1 26))` and `(assert (x2 65))`. Then the tester initiates the testing operation. The expert



(a) Testing the specification.

(b) Testing Eq 2.

Figure 5.5: Testing via array graphs: **T** and **F** represent the truth symbols, **TRUE** and **FALSE**, respectively.

system, then, invokes the rules and generates the temporal and final outputs via the array graph. Figure 5.5-(a) shows the results of the testing.

He may want to perform the I/O testing of an individual equation, `Eq 2` of Figure 2.4, Chapter 2 is performed as illustrated in Figure 5.5-(b):

(1) The human tester sets the mode of the testing as "I/O TESTING".

(2) The human tester clicks the equation node, `Eq 2`, on the displayed array graph using a mouse. It means he wants to test the equation. Then, the expert system recognizes its input and output nodes.

(3) The input nodes, `x1` and `x2`, are shaded by the expert system to denote the status that their input values are needed for the testing.

(4) The user provides test input values for the equation.

(5) The values of the local subscript of the equation (number of test execution of the equation) may be specified by the tester. He positions the cursor next to the subscript and enters a value. Otherwise, the system executes the equation until it terminates.

(6) The expert system executes the specification by firing the rules. The edges (= paths) of the array graph that were visited during the execution are marked to denote the coverage of the testing with the input data.

(7) The output values must be printed vis the output data node as shown in Figure 5.5-(b).

In this particular example, the testing stops after the system outputs `26` for y2(1). Notice that the equation, `Eq 2`, is in fact translated into two rules, `rule3` for `Eq 2-1` and `rule4` for `Eq 2-2`. Since the precondition of `rule3` holds, the value of `y2(1)` can be computed immediately. It follows that the new fact, `(y2 1 26)` is asserted. However, the expert system cannot proceed because the value of `y1(1)` is not defined, i.e. the part of the precondition of `rule4`, `(y1 ?i ?y1)`, is not satisfied. Thus the system notifies that there is data dependency between `Eq 1` and `Eq 2` and stops the testing as shown in Figure 5.5-(b).

The human tester can also perform the path analysis by selecting multiple equation nodes along data paths that he wants to evaluate. The nodes must form a causal path (or chain). Since the causal chain has its inputs, computation on them and the corresponding outputs, the procedure of the path analysis is similar to that of the I/O testing. The tester may select the equations that make a causal path. The system recognizes the inputs and the output of the causal path. The user provides the input values to evaluate the path. Then the system executes the selected equation and the results are computed. The data and the equation nodes are shaded as they are tested. The edges on the causal path are marked.

The path analysis aims to examine not only the outputs of the causal path but also the execution sequences and the data type transformations. To accomplish such objectives, the expert system must be able to do symbolic processing. It requires more rules of manipulating variables and equations of a MODEL specification. We leave it as a future research project and concentrate on the evaluation of MODEL equations using CLIPS rules.

Since the coverage of the test data set used in the path analysis is important, the analyzed causal path of the array graph is marked by "high-lightening" with a different color or by thicker lines as shown in Figure 5.5. The marked graph will become a graphical representation of the test report.

The results of the testing are documented. For the I/O testing, test inputs and their corresponding output are reported for each equation that is tested. The graphical form of the report for the path analysis includes the textual report of testing results and the array graphs each of which specifies the analyzed causal path.

Chapter 6

INTERACTIVE HETEROGENEOUS REASONING FOR PROGRAM VERIFICATION

6.1 Introduction

The formal verification method allows us to prove the reliability of programs with a high degree of confidence in the sense that it actually constructs mathematical proofs [Dij81]. On the other hand, it has been argued that the methodology has not been successfully applied to the real world programs due to the following reasons [Bro87, Den91]:

(1) A human tester of programs, who actually performs the formal verification of the programs, is required to be highly trained in both mathematics and software engineering.

(2) Most real world programs are not "mathematic-prone" (easy to model and construct mathematical proofs). The formal verification has not been very successful for such programs.

(3) Due to the complexity of the formal method, a large-scale system cannot be efficiently verified.

We propose to solve those problems by writing formal specifications in an equational language and adopting software tools of supporting the verification of the specifications. First, we deal with formal specifications written in an equational language, MODEL. We can mechanically translated the MODEL specifications into their equivalent codes in procedural programming languages such as Ada and C [Lu81, PLGS88, PGLS90]. Since a MODEL specification consists of a set of equations and only basic algebraic

manipulation techniques are needed in verifying its correctness as discussed in Chapter 2, the human tester may not need to be a highly trained logician. Secondly, MODEL equations are powerful enough to describe any real world programs. It is capable of declaring and managing complex data structures such as multi-dimensional arrays and structured variables [MOD89]. At the same time, we can view them as a set of algebraic expressions that uniquely define the values of MODEL variables. Thus the proposed verification system is capable of verifying programs in the real world. Finally, the proposed system performs not only program verification but also software testing as discussed in Chapter 5. For a large-scale program, the software testing is mainly performed and the security-critical part of the program is verified using the formal verification method. Also, the MODEL compiler provides a method of static checking [Lu81, PP83, SLPP84, SP88] on the specification as discussed in Chapter 4. Thus the different software tools complementarily work together.

The proposed system uses both the graphical (array graphs) and the textual representations (equations) of the specifications during the verification. The array graphs serve as a graphical user interface of the verification system. An equational reasoning system (called *symbolic manipulator*) based on the equivalence laws and the inference rules proposed in Chapter 2 is a work-horse of the verification system.

Heterogeneous reasoning is defined as a process of reasoning where valid inference proceed from information represented in more than one form [BE90a, BE90b]. The importance of the heterogeneous reasoning in general is argued in [BE90b] and its theoretical framework is introduced in [BE90a]. We view inference as information extraction. That is, inference is a process of deriving implicit information from a given set of explicit information. Note that information can be expressed in multiple formats such as natural languages, formal languages, graphs, diagrams etc. As argued in [BE90b], such information in various formats are very useful in valid reasoning by human beings. Thus we propose to utilize such capability in program verification. Fortunately, we have a tool for representing information in various ways such as graphics and texts, namely a computer. For the heterogeneous reasoning for MODEL specification, an array graph serves as a graphical representation for easy perception of the specification and MODEL equations provides more abstract information. As a result, the proposed system is an interactive inference engine based on the heterogeneous reasoning on array graphs and equations. It is called the interactive heterogeneous reasoning system.

The methodology of the interactive heterogeneous reasoning is described in Section 6.2. It includes assumptions for the verification system and the protocol of the reasoning using the system. The sample specification of the gcd problem is used in demonstrating how the system works. It is illustrated in Section 6.3.

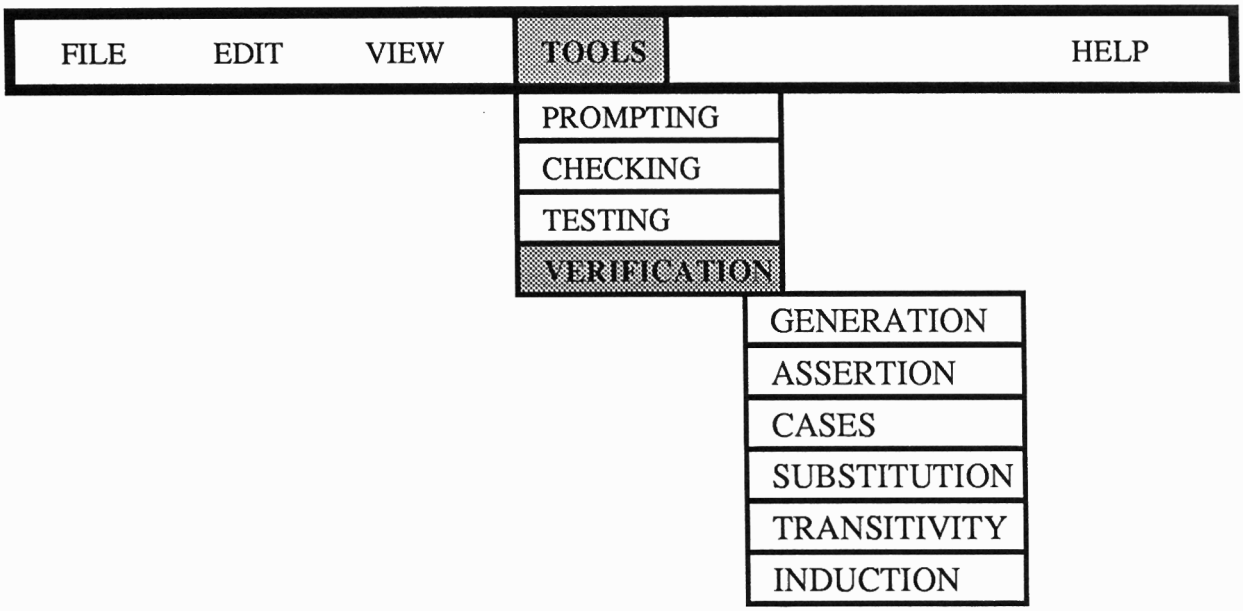


Figure 6.1: Extended pull-down menu for VERIFICATION.

6.2 Interactive Heterogeneous Reasoning

In the interactive heterogeneous reasoning for verifying a MODEL specification, the followings are assumed:

(1) We only deal with a MODEL specification that terminates. Every equation in the specification must terminate. Every variable of each equation must be defined. Thus the symbolic manipulator can algebraically simplify MODEL variables and expressions based on the equivalence laws and the inference rules discussed in Chapter 2.

(2) A human tester must be able to set up his "plan" to prove the properties of the MODEL specification. The properties to be proven are specified as proof goals and subgoals.

(3) An array graph (used as a graphical user interface) is displayed on the array graph window while a set of equations (a MODEL specification being analyzed) is presented on the text window. The human tester can understand and interactively verify the proof goals and the subgoals set up for the specification via the graphical user interface. He uses the symbolic manipulator for simplifying algebraic expressions.

The interactive heterogeneous reasoning system offers a set of operations for the verification of MODEL specifications. Each operation can be invoked by selecting it using a pull-down menu. It is shown in Figure 6.1.

The operation of "GENERATION" aims to find an algebraically simplified expression (with respect to the algebraic laws in Chapter 2) for a designated output variable in terms of designated input variables. A human tester is responsible for designating the variables properly. The symbolic manipulator generates an equation for the output variable in terms of the designated input variables. That is, the output variable becomes the LHS variable of the generated equation while the RHS expression of the generated equation is a function of the input variables. If the symbolic manipulator always derives satisfiable equations from the given set of equations (it is sound), we can use the generated equations as proven assertions (or theorems) for further reasoning. The human tester can initiate the "ASSERTION" operation from the pull-down menu and makes the derived equation be a valid assertion.

When a conditional expression is evaluated, it is necessary to break the expression into cases. The "CASES" operation enables the tester to separate the reasoning procedure according to different cases.

As discussed in Chapter 2, the symbolic manipulator utilizes three inference rules: substitution, transitivity and induction. The human tester dictates what rule must be applied during the interactive heterogeneous reasoning. It is performed by selecting the operations from the pull-down menu and picking up expressions to be manipulated by the designated rule. If the tester chooses the "SUBSTITUTION" operation from the menu in Figure 6.1, he means to replace some variables by their corresponding expressions. Next, he points the variable that he wants to replace using a mouse. He also chooses an expression to be substitute for the variable. The expression might be either the generated one or the one directly from the equations of the specification. The "TRANSITIVITY" operation needs two equations which share one variable or an expression. As an output of the operation, a new equation is formulated. Finally, the tester can construct an inductive proof for a theorem using the "INDUCTION" operation. He supplies a base case and an induction step to get a theorem proven by the symbolic manipulator.

6.3 Example

The protocol of the interactive heterogeneous reasoning is illustrated using the example of computing a gcd of two integers presented in Figure 2.4, Chapter 2. As assumed before, the following proof goals and their subgoals are given:

Goal I. $z = gcd(x1, x2)$

Subgoal 1. $gcd(x1, x2) = gcd(y1(1), y2(1))$

Subgoal 2. $gcd(y1(i), y2(i)) = gcd(y1(i-1), y2(i-1))$

Subgoal 3. $z = gcd(y1(SIZE.y1), y2(SIZE.y1))$

Goal II. $SIZE.y1 < finite\_val$

Subgoal 1. $max(y1(i-1), y2(i-1)) - max(y1(i), y2(i)) \geq 1$

Subgoal 2. $SIZE.y1 < max(x1, x2)$

A human tester may have the following plan to prove Subgoal 1:

P-1. Express the first elements of the array variables, $y1(1)$ and $y2(1)$, in terms of the input variables, $x1$ and $x2$.

P-2. Derive the LHS expression, $gcd(x1, x2)$, from the RHS expression, $gcd(y1(1), y2(1))$.

P-3. Conclude $gcd(x1, x2) = gcd(y1(1), y2(1))$.

The first step, P-1, can be achieved by the "GENERATION" operation. The operation is selected first from the pull-down menu illustrated in Figure 6.1. Then, the human tester chooses a target variable, which becomes the LHS variable of the generated equation, from the displayed array graph using a mouse. In this particular example, the variable, $y1(1)$, is selected first. By reviewing the displayed array graph and the equations, the human tester can figure that the value of $y1(1)$ must be defined by the equation, `Eq 1`, and its inputs must be $x1$ and $x2$. So he designates its source variables, $x1$ and $x2$. Those variables are marked in the displayed array graph. The system then invokes the symbolic manipulator. Thus, the simplified equation, $y1(1) =$ **IF** $x1 > x2$ **THEN** $x1$ **ELSE** $x2$, is produced through the text window. It ends one cycle of the operation.

Since the derived expression is conditional and not fully simplified, the tester breaks it into two cases using the "CASES" operation: $x1 > x2$ and $x1 \leq x2$. The symbolic manipulator then evaluates the expression and derives equations, $y1(1) = x1$ and $y1(1) = x2$, respectively, as shown in Figure 6.2.

To get an expression for $y2(1)$ in terms of $x1$ and $x2$, the "GENERATION" operation is invoked again. The human tester marks the variable, $y2(1)$, as a target variable. The input variables, $x1$ and $x2$, are chosen. The symbolic manipulator works. The equation, $y2(1) =$ **IF** $x1 > x2$ **THEN** $x2$ **ELSE** $x1$, is produced as an output on the text window. Like the case for $y1(1)$, the "CASES" operation is needed as illustrated in Figure 6.2.

The generated equations can be new assertions (or theorems) for the program verification. The tester may want to modify them into simpler form. He simplifies the results from the "GENERATION" operation and stipulates them as new assertions. It is performed by the "ASSERTION" operation. As shown in Figure 6.3, $y1(1) = x1, y2(1) = x2$ when $x1 > x2$ and $y1(1) = x2, y2(1) = x1$ when $x1 \leq x2$. They are stored in the knowledge base of the symbolic manipulator, where other equations are also contained.

The second step, P-2, is performed through the "SUBSTITUTION" operation and the "CASES" operation. Since variables $y1(1)$ and $y2(1)$ have different values according to the values of $x1$ and $x2$, Subgoal 1 must be proven for the different two

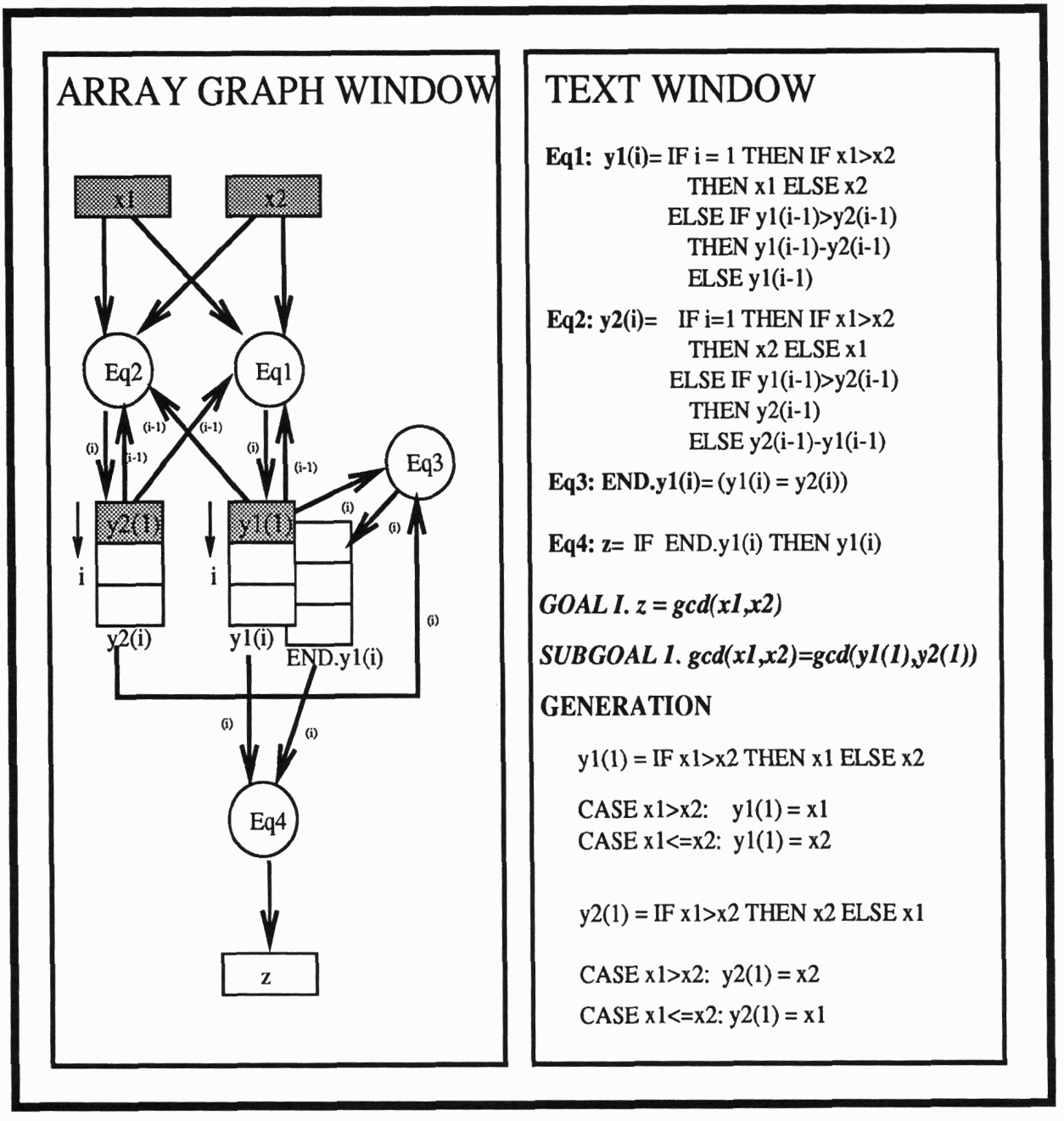


Figure 6.2: Generation.

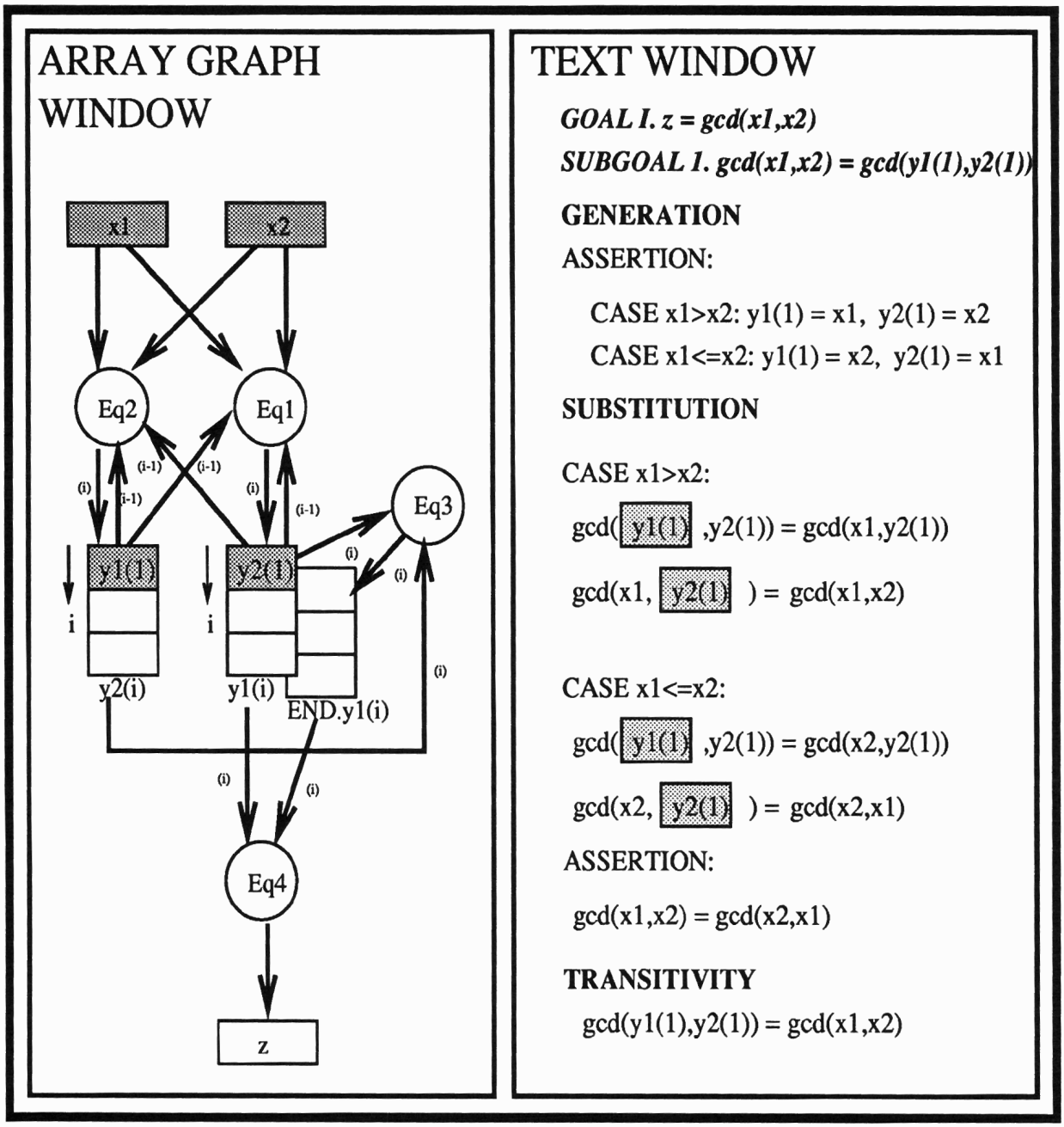


Figure 6.3: Substitution.

cases. Suppose $x1 > x2$. The human tester can designate variable $y1(1)$, which is equal to $x1$, in the expression as shown in Figure 6.3. The symbolic manipulator, then, searches its knowledge base to get the assertion for the variable, namely $y1(1) = x1$. Next, the expression, $gcd(x1, y2(1))$, is generated by the symbolic manipulator. It must be equal to the expression, $gcd(y1(1), y2(1))$. Thus the equation, $gcd(y1(1), y2(1)) = gcd(x1, y2(1))$, is displayed in the text window and stored the knowledge base. Next, the human tester replaces the variable, $y2(1)$, by $x2$ by marking variable $y2(1)$. Then the symbolic manipulator substitutes $x2$ for variable, $y2(1)$. Therefore, we get the equation, $gcd(x1, y2(1)) = gcd(x1, x2)$ as shown in Figure 6.3. It partially proves Subgoal 1. Secondly, the case of $x1 \leq x2$ is checked. Similarly, we can get $gcd(y1(1), y2(1)) = gcd(x2, x1)$. As shown in Figure 6.3, the tester may use a new assertion $gcd(x1, x2) = gcd(x2, x1)$ which is derived from the requirement assertions of Euclid's algorithm for the gcd problem. Then, the transitivity rule is applied to conclude that $gcd(y1(1), y2(1)) = gcd(x1, x2)$ for the case.

The third step, P-3, concludes that that Subgoal 1 is proven.

Consider Subgoal 2, $gcd(y1(i), y2(i)) = gcd(y1(i-1), y2(i-1))$. It has to be proven by the following two cases: $y1(i-1) > y2(i-1)$ and $y1(i-1) \leq y2(i-1)$. As shown in Figure 6.4, the requirement assertion is used to prove Subgoal 2. Let v be $y1(i-1)$ and w be $y2(i-1)$. It is the first case, $y1(i-1) > y2(i-1)$. The "SUBSTITUTION" operations and the "GENERATION" operation lead the conclusion. Similarly, the second case can be proven. The rest of the proof goals and subgoals can be proven in the similar fashion.

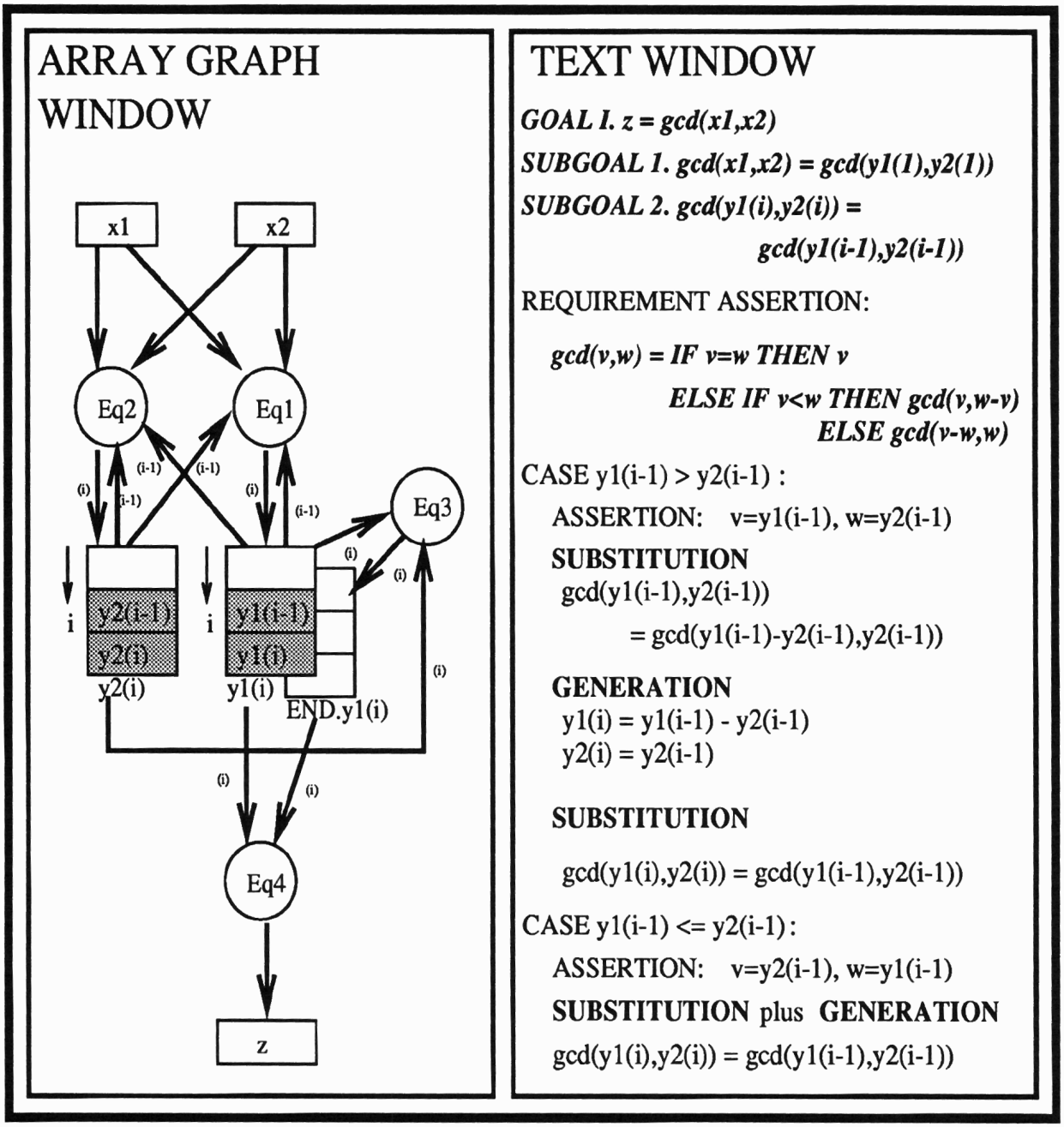


Figure 6.4: Proof of Subgoal 2.

Bibliography

[Arv82] T. A. Arvind. Data flow systems. *Computer*, February 1982.

[AS90] P. J. Asente and R. R. Swick. *X Window System Toolkit.* Digital Press, Bedford, Massachusetts, 1990.

[AW76] E. A. Ashcroft and W. W. Wadge. Lucid — a formal system for writing and proving programs. *SIAM Journal of Computing*, 5(3):336–354, September 1976.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of ACM*, 20(7):519–526, July 1977.

[Bac78] John W. Backus. Can programming be liberated from the Von Neumann style: a functional style and its algebra of programs. *Communications of ACM*, 21(8):613–640, 1978.

[BE90a] Jon Barwise and John Etchemendy. Information, infons and inference. In R. Cooper, K. Mukai, and J. Perry, editors, *Situation Theory and its Applications*, University of Chicago Press, 1990.

[BE90b] Jon Barwise and John Etchemendy. Visual information and valid reasoning. In W. Zimmerman, editor, *Visualization in Mathematics*, pages 9–24, MAA, 1990.

[Bei90] B. Beizer. *Software Testing Techniques — Second Edition.* Van Nostrand Reinhold Company, New York, New York, 1990.

[BGM90] R. S. Boyer, M. W. Green, and J. S. Moore. The use of a formal simulator to verify a simple real-time control program. In W.H.J. Feijen, A.J.M. van Gasteren, D. Greis, and J. Misra, editors, *Beauty Is Our Business: A Birthday Salute to Edsger W. Dijkstra*, pages 54–66, Springer-Verlag, New York, New York, 1990.

[BM81] R. S. Boyer and J. S. Moore. A verification condition generator for FORTRAN. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, pages 9–101, Academic Press, London, England, 1981.

[BMSW90] L.M. Burbs, A. Malhotra, G.H. Sockut, and K.Y. Whang. *AERIAL: Ad hoc Entity-Relationship Investigation and Learning.* Research Report RC 16186, IBM T.J.Watson Research Center, October 1990.

[Bro87] F. P. Brooks, Jr. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[Bru89] J. Bruno. *Analyzing Conditional Data Dependencies in an Equational Language Compiler.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, December 1989.

[CCL91] Marina Chen, Young-il Choo, and Jingke Li. Crystal: theory and pragmatics of generating efficient parallel code. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 255–308, ACM Press, 1991.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[Cha90] S. K. Chang. *Visual Languages and Visual Programming.* Plenum Press, 1990.

[Cla76] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, September 1976.

[CPS90] R. Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: a semantics based tool for the verification of concurrent systems. 1990. manuscript.

[Dec90] *Guide to DECdesign.* digital equipment corporation, Maynard, Massachusetts, May 1990.

[Den91] Peter J. Denning. Beyond formalism. *American Scientist*, 79:8–10, January-February 1991.

[Dij81] E. W. Dijkstra. Why correctness must be a mathematical concern. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, pages 1–8, Academic Press, London, England, 1981.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[DMMP87] R. A. DeMillo, W. M. McCracken, R. J. Martin, and J. F. Passafiume. *Software Testing and Evaluation.* Benjamin/Cummings Publishing Co., Menlo Park, California, 1987.

[For82] C. L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[Gal86] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving. Harper and Row Computer Science and Technology Series*, Harper and Raw Publishers Inc., New York, New York, 1986.

[Ge89] Xiang Ge. *An Intelligent Mathematical Modeling System.* PhD thesis, University of Pennsylvania, 1989.

[GH88] D. Gelperin and B. Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, 1988.

[Gil76] W. J. Gilbert. *Modern Algebra with Applications.* John Wiley and Sons, 1976.

[GJ90] D. Gelernter and S. Jagannathan. *Programming Linguistics.* The MIT Press, Cambridge, Massachusetts, 1990.

[GP89] Xiang Ge and Noah S. Prywes. Reverse software engineering of concurrent programs. 1989. manuscript.

[GR89] J. C. Giarratano and G. Riley. *Expert Systems: Principles and Programming.* PWS-KENT Publishing Company, 1989.

[Gri81] David Gries. *The Science of Programming. Text and Monographs in Computer Science*, Springer-Verlag, New York, New York, 1981.

[Ham88] R. Hamlet. Special section on software testing. *Communications of the ACM*, 31(6):662–667, June 1988.

[Har88] D. Harel. On visual formalisms. *Communications of the ACM*, 514–530, May 1988.

[HM85] M. C. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of ACM*, 32(1):137–161, January 1985.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[How86] W. E. Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, SE-12(10):997–1005, October 1986.

[How87] W. E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill Inc., New York, New York, 1987.

[Hud89] Paul Hudak. Conception, evolution and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[Hud91] Paul Hudak. Para-functional programming in Haskell. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 105–158, ACM Press, 1991.

[Kin76] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[Kro87] F. Kröger. *Temporal Logic of Programs.* Volume 8 of *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, New York, New York, 1987.

[Lam83] L. Lamport. What good is temporal logic? In *Proceedings IFIP Congress, Paris*, pages 657–668, North-Holland, Amsterdam, Holland, 1983.

[Lan66] E. Landau. *Foundations of Analysis: The Arithmetic of Whole, Rational, Irrational and Complex Numbers.* Chelsea Publishing Co., New York, New York, third edition, 1966. translated by F. Steinhardt in English.

[Lin88] P. A. Lindsay. A survey of mechanical suppport for formal reasoning. *Software Engineering Journal*, 3–27, January 1988.

[LP90] Evan Lock and Noah S. Prywes. *Software Engineering Environment for Parallel/Concurrent Programs on a Computer Network.* Technical Report, Computer Command and Control Company, 2300 Chestnut Street, Philadelphia, PA 19103, 1990.

[Lu81] K. S. Lu. *Program Optimization Based on a Non-Procedural Specification.* PhD thesis, University of Pennsylvania, 1981.

[Man74] Z. Manna. *Mathematical Theory of Computation. McGraw-Hill Computer Science Series*, McGraw-Hill Book Co., New York, New York, 1974.

[Met91] William Mettrey. A comparative evaluation of expert system tools. *Computer*, 19–31, February 1991.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[MOD89] *The MODEL Compiler Usage and Reference Guide — Non-Procedural Programming for Non-Programmers.* Computer Command and Control Company, 2300 Chestnut Street, Philadelphia, PA 19103, 1989.

[MP81] Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In *The Correctness Problem in Computer Science*, pages 215–273, Academic Press, London, England, 1981.

[Mye88] B. A. Myers. *The State of Art in Visual Programming and Program Visualization.* Technical Report CMU-CS-88-114, Carnegie Mellon University, February 1988.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[PGLS90] Noah S. Prywes, Xiang Ge, Insup Lee, and Mitchel Song. *Procedural to Equational Language Translation.* Technical Report Contract AFSOR-88-0116, Department of Computer and Information Science, University of Pennsylvania, 1990.

[PLGS88] Noah S. Prywes, Insup Lee, Xiang Ge, and Mitchel Song. *Reverse Software Engineering.* Technical Report MS-CIS-88-99, Department of Computer and Information Science, University of Pennsylvania, 1988.

[PLK91] Noah S. Prywes, Insup Lee, and Jee-In Kim. Extracting rules from software for use in a knowledge base of an expert system. August 1991. Prepared for Workshop at Rome, NY in October, 1991.

[PP83] Noah S. Prywes and A. Pnueli. Compilation of nonprocedural specifications into computer programs. *IEEE Trans. on Software Engineering*, SE-9(3):267–279, May 1983.

[RW85] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[Set89] R. Sethi. *Programming Languages: Concenpts and Constructs.* Addison-Wesley, 1989.

[SGN88] R. W. Scheifler, James Gettys, and Ron Newman. *X Window System — C Library and Protocol Reference.* Digital Press, Bedford, Massachusetts, 1988.

[Shi91] Sun-Joo Shin. *Valid Reasoning and Visual Representation.* PhD thesis, Stanford University, August 1991.

[SLPP84] B. Szymanski, Evan Lock, A. Pnueli, and Noah S. Prywes. On the scope of static checking in definitional languages. In *Proceedings of the ACM Annual Conference*, pages 197–207, San Francisco, California, October 1984.

[SP88] B. K. Szymanski and Noah S. Prywes. Efficient handling of data structures in definitional languages. *Science of Computer Programming*, 10:221–245, 1988.

[Szy91] B. K. Szymanski. EPL — parallel programming with recurrent equations. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 51–104, ACM Press, 1991.

[TB86] T. H. Taylor and R. P. Burton. An icon-based graphical editor. *Computer Graphics World*, 77–82, October 1986.

[Wey86] E. J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, SE-12(12):1128–1138, December 1986.

[Wey90] E. J. Weyuker. The cost of data flow testing: an empirical study. *IEEE Transactions on Software Engineering*, SE-16(2):121–128, February 1990.