(2)

SRI International

A002: FINAL REPORT • May 18, 1991

THE PEGASYS ENVIRONMENT FOR GRAPHICAL DOCUMENTATION OF LARGE PROGRAMS

Prepared by:

Mark Moriconi, Director
Computer Science Laboratory

SRI Project 2684
Contract No. N00014-86-C-0075

Prepared for:

Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5000
Attn: Mr. James G. Smith
Applied Research and Technology
Code 1211

Visual Specification and Reasoning in PegaSys

Mark Moriconi

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, California 94025

May 16, 1991

Abstract

This is the final report for ONR contract number N00014-86-C-0775. It describes the PegaSys languages, methodology, and techniques for specifying and reasoning about system structures. PegaSys supports both visual and textual specification, where pictures are intuitive representations of logical assertions written in the textual language. To mitigate the problems of scale, the PegaSys methodology supports both horizontal and vertical hierarchies and provides for user-defined abstractions in specifications. Since PegaSys is based on logic, it can reason about designs and programs. For example, it can prove (automatically) that a structural design hierarchy is correct and find (automatically) a conservative approximation of the semantic effects of changes to programs. These advances will allow the PegaSys environment to be considerably more powerful than the CASE tools currently used in industry to develop large software systems.

The first part of this report presents a scenario that illustrates the basic ideas behind the PegaSys languages and methodology. The underlying logic is a decidable subset of Ehdm, a state-of-the-art formal specification language developed in the Computer Science Laboratory. The second part of the report presents the details of our technique for deducing the effects of changes to a program. (Changes to a design are not handled.) The new material presented in this report is not implemented in PegaSys at this time.

Contents

List of Figures

Chapter 1

Sample PegaSys Scenario

1.1 Introduction

The PegaSys language and methodology is intended for use in the visual structuring of large software designs. A system is partitioned by a hierarchy of linked diagrams, representing abstract system requirements as well as concrete implementation structures. The exact relationship between levels in a hierarchy is specified explicitly.

The PegaSys system has several unique features not present in other CASE technologies.

- **Refinement hierarchies.** A PegaSys system specification is a collection of diagrams linked together "horizontally" to form complete design levels and "vertically" to construct refinements that bring the existing structures closer to an efficient, practical realization. Most CASE methodologies support the development of horizontal hierarchies (sometimes called leveling). None support true vertical hierarchies, and we consider this to be one of PegaSys' unique strengths. Vertical hierarchies are crucial in system design, implementation, and evolution because of the inherent differences between abstract and concrete realizations of the same system.
- **Multiple representations.** Diagrams are supposed to make it easy to see relationships among objects. If this clarity is lost, the advantage of diagrams is lost. An inherent problem with diagramming techniques is that diagrams can become very cluttered and too large for a single page or monitor. Diagram decomposition (leveling) can help to some degree, but it also can cause a loss of context because of the large number of small pieces that must be related.

The PegaSys methodology increases comprehensibility by providing for a visual and a textual representation of the same specification. A diagram is seen as a visual representation of a more compact logical assertion. The textual assertion can contain more information than the diagram itself. In fact, multiple diagrams can be associated with a single assertion, each diagram providing a different "view" of the assertion.

- **Logical precision.** Pictures are intuitive representations of logical assertions, allowing inferences to be drawn about an individual picture or a collection of pictures. For example, it is possible to determine whether a picture at a given level in a design hierarchy is a correct refinement of a higher-level picture. This kind of analysis cannot be added easily to existing CASE tools.

- **Flexibility.** Diagrams can contain new concepts that are defined in terms of the predefined primitives. Existing approaches to structural design provide a small, fixed set of primitive relations, and it is not possible to build up new relations from the primitives. As a consequence, designs often are too concrete and not truly hierarchical.

- **Unified model.** PegaSys provides a single, unified design model. Existing methodologies use two or more separate models accompanied by various mechanisms for relating the models. For example, the Hatley/Pirbhai design technique involves data flow diagrams, control flow diagrams, and architectural diagrams, but no clear methodology is given for connecting the models.

1.2 Overview of the PegaSys Methodology

A structural specification of a hardware/software system consists of a collection of linked diagrams. An individual diagram denotes objects and interrelationships among the objects. Active objects, such as processes and subprograms, accept input and produce results, while passive objects, such as types and variables, represent the data manipulated by active objects. Both active and passive objects can be passed as arguments to active objects.

Objects are grouped together into modules. Modules can be used to hide implementation details through selective exporting of names. A module can be *generic* in that it can be parameterized by types and constants. A generic module can be instantiated to form an unparameterized module, which is referred to as a *module instance.* Generic modules provide an effective mechanism for design reuse.

Explicit links between diagrams are used to build two kinds of hierarchies, each of which is useful in structuring large systems. Typically, a structural description of a system starts with abstract functional structures, such as dataflow diagrams, which ultimately are refined into detailed, implementation-level structures. The links between diagrams make explicit the intended relationships between the diagrams.

1.2.1 Hierarchies

Diagrams are linked together "horizontally" to form complete design levels and "vertically" to construct refinements. A *horizontal hierarchy* is a set of diagrams that elaborate some large diagram from a collection of smaller diagrams, in much the same way that a large program is built from smaller program units. A *vertical hierarchy* refines or implements a diagram at one level of abstraction in terms of diagrams containing more concrete objects and interconnections. The intent is not to add new structures, but to bring the existing ones closer to an efficient, practical realization.

The PegaSys methodology allows precise specification of the mapping between levels in a vertical hierarchy. The mapping describes how to interpret the concepts of a given level in terms of those of a more abstract level in a vertical hierarchy. More specifically, the objects and interconnections (relations) at a given level in a vertical hierarchy must be placed in (one-to-many) correspondence with the the objects and relations at the next lower level.[1]

Every horizontal hierarchy (i.e., each level in the vertical hierarchy) is expected to be complete with respect to the given level of detail. This is important so that we can conclude that, if a connection is not specified, it does not exist. For example, if two processes have no direct connection in a diagram, we want to be able to conclude that they do not pass messages to each other. This would be invalid unless if we had not assumeed that all inter-process communication was specified.

In a vertical hierarchy, there can be three kinds of refinements: dependency refinement, active-object refinement, and passive-object refinement. Suppose that two processes communicate by message passing. A dependency refinement may implement the concept of message passing by the reading and writing of a shared variable. An active-object refinement may implement a process by means of several sequential subprograms. A passive-object refinement may specify the exact structure of messages.

[1]The concept of vertical refinement is somewhat similar to the concept of model in logic. In particular, the mapping from one level in a vertical hierarchy to the next lower level is analagous to an interpretation that maps a logical theory to its model.

1.2.2 Two-Tiered Representation

The PegaSys methodology takes into account that diagrams can become very cluttered, difficult to understand, and sometimes ineffective at representing a design decision. In particular, PegaSys provides a two-tiered representation of a system specification. At one level are logical diagrams; at the other level is the textual assertion that the diagram depicts. Multiple diagrams can be used to provide different views of the same assertion.

The textual description of a system can contain more information that the corresponding diagram(s). In fact, a diagram should be viewed as a combination of graphics and text. For example, it may be easier to specify a data structure in text. If that is the case, it should be possible to enter the specification textually and not be forced to develop suitable icons. In general, a PegaSys user should be able to enter specifications almost entirely graphically, entirely textually, or in some combination of both.

Any textual description of a system should be written in a language that is expressive and suitable for effective communication among its human users. PegaSys provides a language that attempts to meet these requirements.

Underlying the textual language is a logic that is more austere and suitable for mechanical analysis by computer. However, the PegaSys user need not be aware of this language.

1.3 Developing Specifications in PegaSys

In this section, we will develop a simple specification that is represented both diagrammatically and textually. We will introduce the PegaSys language and methodology by means of a simple example, namely, a vending machine.

The vending machine accepts coins and a product selection from a customer, dispenses the product to the customer if the payment is sufficient, and returns the correct change if the deposited amount is too much. If the product is not available, all coins are returned. We do not want the customer to make a selection without entering coins, and we prohibit a product from being dispensed before the customer's selection and payment have been validated.

Before we begin to design the vending machine, it is important to understand that we will not specify what the machine is intended to do. Instead, we specify the structure of the vending machine.

To get started, we draw a so-called context diagram, shown in Figure 1.1, that shows the inputs and outputs of the vending machine. In particular, the diagram shows four input data flows and four output data flows; the source, sink, and the vending machine itself are modeled as concurrent processes. The numbers in the bubbles are bookkeeping aids and do not affect the meaning of the diagram.

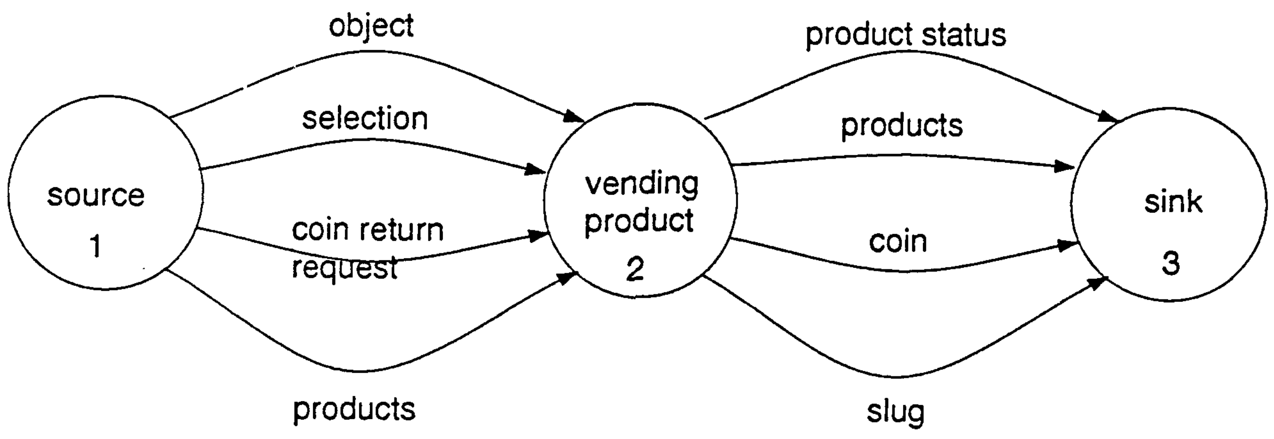

Figure 1.1: Vending Machine Interface

The context diagram is intended to provide an intuitive yet precise specification of system dependencies. These dependencies must be implemented at lower levels in the design of the vending machine. If we want to know exactly what the context diagram says, we must look at the underlying textual specification, only part of which is depicted in the diagram.

We will develop the textual specification in pieces. First, we introduce several value domains, or types, that are referred to in the diagram.

```
coin, slug: TYPE
object: TYPE = coin | slug
selection, coin_return_request, products, product_status: TYPE
```

These declarations introduce seven new types. The structure of type `object` is specified; namely, it is required to be either a coin or a slug. However, the structure of the other types is left unspecified for the moment.

In these declarations, `TYPE` is a keyword. By convention, PegaSys keywords are written in uppercase, but they can be written in any case. For example, `TYPE`, `Type`, `type`, and even `tYpE` are all the same keyword. Identifiers, however, are case-sensitive: `coin` and `Coin` are different identifiers. PegaSys identifiers consist of a letter (upper or lower case), followed by any sequence of letters, digits, and underscore characters. As with many programming languages, adjacent PegaSys keywords and/or identifiers must be separated from each other by spaces. Unlike most programming languages, PegaSys declarations and expressions are *not* terminated by semicolons.

Next, we need to declare the "signatures" of the active objects and the predicates that we intend to use. The "source" process takes no value as input and produces values of four different types as output. In PegaSys this is expressed as

```
source: PROCESS [ -> object, selection, coin_return_request, products]
```

The other two processes are declared in a similar manner.

```
vending_product: PROCESS [object, selection, coin_return_request, products
                          -> product_status, products, coin, slug]
sink:    PROCESS [product_status, products, coin, slug -> ]
```

Dependencies among the declared objects are specified using the predefined predicate

```
dflow: FUNCTION [process, process, type -> BOOLEAN]
```

which is true provided the first process passes values of the specified type to the second process. (A predicate is simply a function with return type `BOOLEAN`.) In particular, we have

```
in: ASSERT dflow(source, vending_machine,
                {object, selection, coin_return_request, products})
out: ASSERT dflow(vending_machine, sink, {product_status, product, coin, slug})
```

These flow relations are called assertions because they make claims about the implementation at the next level in the vertical hierarchy. The labels in and out allow us to refer to the associated relations by name. The set notation in

```
in: ASSERT dflow(source, vending_machine,
                {object, selection, coin_return_request, products})
```

is a convenient shorthand for

```
in1: ASSERT dflow(source, vending_machine, object)
in2: ASSERT dflow(source, vending_machine, selection)
in3: ASSERT dflow(source, vending_machine, coin_return_request)
in4: ASSERT dflow(source, vending_machine, products)
```

Having specified the objects and relations in the context specification, we have only one thing left to do. We must decide whether any of the three processes (bubbles in the diagram) are just convenient abstractions that are *not* intended to be represented directly in the implementation of the vending machine. You can think of such bubbles as bookkeeping mechanisms for keeping track of sets of lower-level bubbles.

In our example, the `vending_product` process is not intended to be implemented directly. This is not apparent in the diagram, but it must be made explicit in the textual representation of the diagram. The clause

```
REPLACE vending_product WITH vending_machine [ALL]
```

says that the vending-product bubble will be *physically* replaced by an object called `vending_machine` with the same parameters as `vending_product`. The keyword `ALL` saves us the trouble of listing the parameters. This view of refinement is analagous to the concept of macro expansion in assembly languages. It is also the view supported by most CASE systems. (Later, we will see an example in which a bubble cannot be replaced by the bubbles that implement it.)

We can now pull all of this together into the PegaSys module called `vm_io` in Figure 1.2. A PegaSys module is rather like a module or package in some modern programming languages: it serves to group related things together into a unit that can be used many times over, and it serves to delimit the scope of identifiers. By default the identifiers declared in a module are not visible outside the module unless they are explicitly "exported." functions and predicates, which are constants of a certain "higher" type). The EXPORTING construct is used to list the identifiers that are to be visible outside the module.

```
vm_io: MODULE

  -- declarations of input/output values

  coin, slug: TYPE
  object: TYPE = coin | slug

  selection, coin_return_request, products, product_status: TYPE

  -- the environment
  source: PROCESS [ -> object, selection, coin_return_request, products]
  sink:   PROCESS [product_status, products, coin, slug -> ]

  -- the product
  vending_product: PROCESS [object, selection, coin_return_request, products
                            -> product_status, product, coin, slug]

  -- logical decomposition
  REPLACE vending_product WITH vending_machine [ALL]

  -- wiring of vm_io objects
  in: ASSERT dflow(source, vending_machine,
                   object, selection, coin_return_request, products)
  out: ASSERT dflow(vending_machine, sink, product_status, products, coin, slug)

END vm_bio
```

Figure 1.2: Textual Specification of Vending Machine Interface

The specification in Figure 1.2 can be derived from the diagram in Figure 1.1, except for the REPLACE statement. We will see several more examples of diagrams that only partially represent the text.

1.4 Horizontal Hierarchies

We now proceed to build a horizontal hierarchy, using the context diagram in Figure 1.1 as the starting point. The hierarchy will be horizontal because there will *not* be an important change in representation. In particular, we further elaborate the dataflow decomposition of the vending machine before making decisions about how to implement it.

1.4.1 External Interfaces: Parameterization and Wiring

The diagram in Figure 1.3 illustrates a dataflow design for a vending machine. The dashed arrows do not denote a specific flow relation; they represent required inputs and computed outputs. For example, the bubble labeled "coin receptacle must be invoked with values of type "object" and it produces values of type "slug". The method of transmission is left unspecified and, in general, there are many ways to transmit values. A specific method of transmission is made explicit when the diagram is integrated into a given context. We avoid overspecification to increase the likelihood that the diagram will be reused.

Let us begin developing the textual representation of the vending machine. The interface types are not declared locally; they are parameters of a generic vending-machine module that encapsulates the entire diagram.

```
vending_machine: MODULE [object, slug, coin, product_status,
                         products, coin_return_request, selection: TYPE]
```

The required local types (i.e., those not visible to clients of the module) are declared without any indication as to their internal structure.

```
sufficient_payment, current_payment, coin_detected, change_due,
status, product_id, product_availability_info, coin_return_status,
customer_selection:  TYPE
```

Next, we declare the processes in the diagram, paying close attention to how each process is intended to be "wired up" to its clients. The keyword IN indicates an input parameter and the keyword OUT indicates an output parameter. Using these conventions, the following signature specifies how to wire the "coin receptacle" process into an external context.

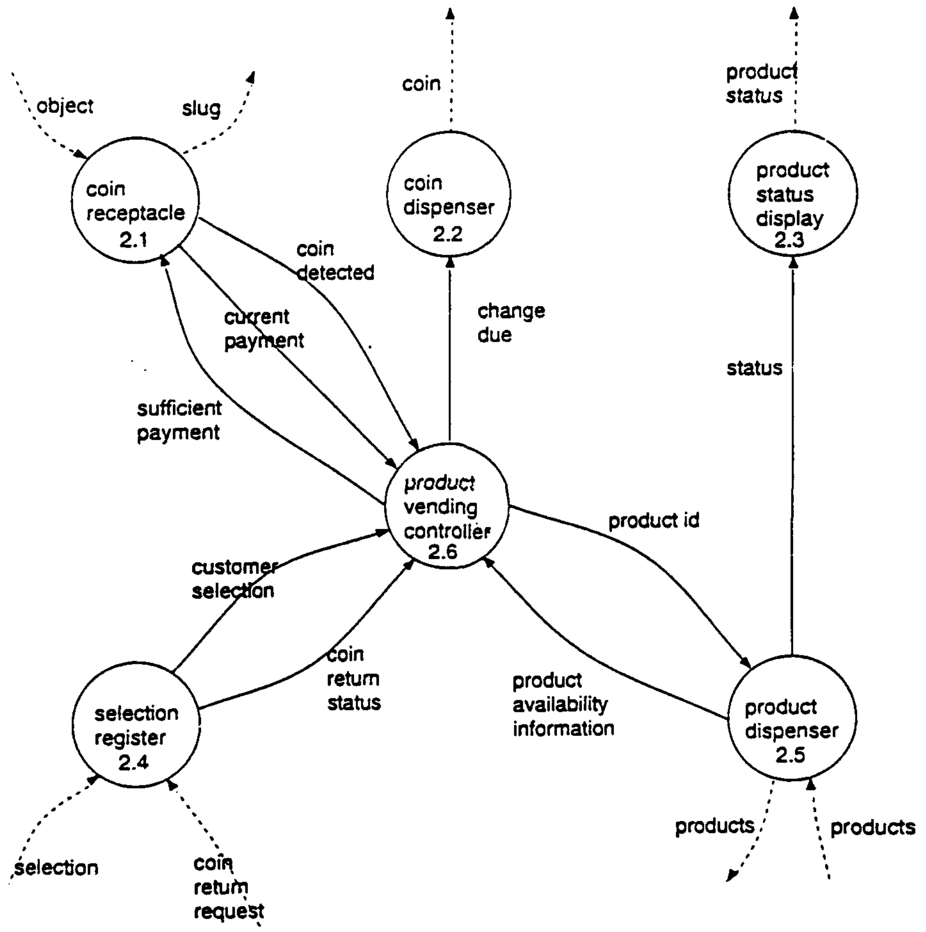

Figure 1.3: Vending Machine

```
coin_receptacle: PROCESS [IN object -> OUT slug, current_payment, coin_detected]
```

The "coin receptacle" process requires an external input of type `object` and produces an external output of type `slug`. The other two outputs are for the internal wiring of the vending machine and are not visible to clients of the vending machine module. In general, there can be many different ways to wire up a process. It is necessary to differentiate internal and external parameters to guarantee that we get the intended wiring. The internal wiring of `vending_machine` is specified using the `dflow` relation.

The "product vending controller" is a logical abstraction that we do not intend to represent directly in an implementation. In particular, we want to replace the associated bubble with a diagram, which we indicate by

```
REPLACE product_vending_controller WITH pvc [ALL]
```

Module `pvc` will be defined next.

The complete textual specification for the vending machine is contained in Figure 1.4. The fact that `vending_machine` is a generic module is not represented in Figure 1.3; neither is the fact that `product_vending_controller` is intended to be eliminated by macro substitution.

1.4.2 Uses Decomposition

A large diagram is built from pieces in two ways: through replacement modules and through used modules. The PegaSys constructs that link modules in these ways are the `REPLACE` and the `USING` clauses, respectively. We have seen two examples of the former where a bubble was replaced by a diagram. The original bubble was more of an organizational device than a design concept. In contrast, used modules perform much the same role as subprograms in a conventional programming language, which are not expanded as macros at the level of the source code. The distinction is important because replaced bubbles need not be implemented in a vertical hierarchy. ·

A used module is contained in the diagram in Figure 1.5. The module labeled "price table manager" is an independent module used to provide mutual exclusion for a shared table that contains the price of each product. The table manager module hides the representation of the table from users of the table. If we allowed the table manager to be expanded as a macro, the internal representation of the table would be visible to clients, thereby violating the principles of information hiding.

The price table manager provides two types and one operation to clients.

```
vending_machine: MODULE [object, slug, coin, product_status,
                         products, coin_return_request, selection: TYPE]

  -- internal value domains

  sufficient_payment, current_payment, coin_detected, change_due,
  status, product_id, product_availability_info, coin_return_status,
  customer_selection:  TYPE

  -- wire into external context

  coin_receptacle: PROCESS
      [IN object -> OUT slug, current_payment, coin_detected]
  coin_dispenser: PROCESS [change_due -> OUT coin]
  product_status_display: PROCESS [status -> OUT product_status]
  product_dispenser: PROCESS
      [IN products, product_id  -> product_availability_info, OUT products]
  selection_register: PROCESS
      [IN selection, IN coin_return_request
        -> customer_selection, coin_return_status]
  product_vending_controller: PROCESS
      [coin_detected, current_payment, product_availability_info,
       customer_selection, coin_return_status
         -> sufficient_payment, change_due, product_id]

  -- logical refinement
  REPLACE product_vending_controller WITH pvc [ALL]

  -- internal wiring

  cr   : ASSERT dflow(coin_receptacle,product_vending_controller,
                      coin_detected, current_payment)
  pd1 : ASSERT dflow(product_dispenser, product_status_display, status)
  pd2 : ASSERT dflow(product_dispenser, product_vending_controller,
                      product_availability_info)
  sr   : ASSERT dflow(selection_register, product_vending_controller,
                      customer_selection, coin_return_status)
  pvc1: ASSERT dflow(product_vending_controller, coin_receptacle,
                      sufficient_payment)
  pvc2: ASSERT dflow(product_vending_controller, coin_dispenser, change_due)
  pvc3: ASSERT dflow(product_vending_controller, product_dispenser, product_id)

END vending_machine
```

Figure 1.4: Textual Specification of Vending Machine

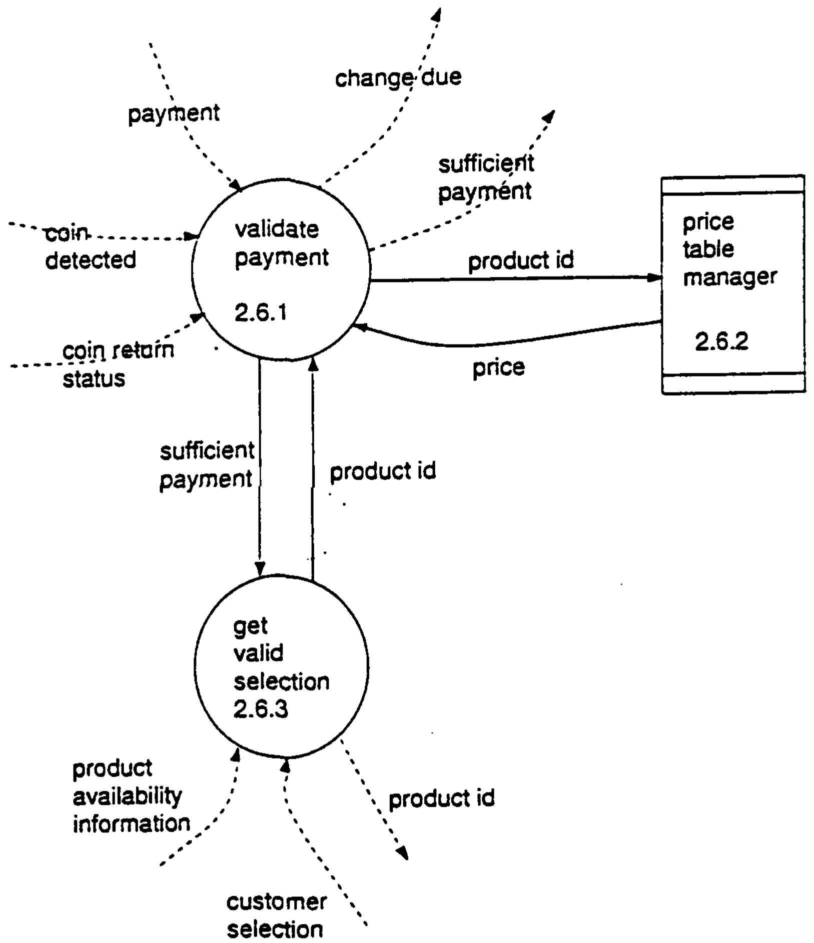

Figure 1.5: Product Vending Controller

```
product_id, price: TYPE
get_price: PROCESS [product_id -> price]
```

Consistent with our style thus far, we have not specified the structure of the types. (An operation to fill the table has been omitted since it is not used in the example.) We will show how the table is represented when we build the vertical hierarchy.

The table manager must also ensure that only certain operations are exported to users.

```
EXPORTING get_price, product_id, price
```

So far, all declared objects are exported, but the price table will be defined later and it will not be exported. The table manager interface specification is contained in Figure 1.6.

```
price_table_mngr: MODULE                -- this will be a monitor
  EXPORTING get_price, product_id, price
                                        -- price table hidden from clients
  product_id, price: TYPE
  get_price: PROCESS [product_id -> price]

END price_table_mngr
```

Figure 1.6: Textual Specification of Price Table Manager

We are now ready to specify the product vending controller. The technique is essentially the same as the one used for the vending machine, with one exception. The controller imports the table manager, so that it can reference the objects exported by the controller.

```
pvc: MODULE [ ... : TYPE]
  USING price_table_mngr

  validate_payment: PROCESS
      [ ...  product_id, price ->  ...  product_id]
  ...
END pvc
```

The ellipses indicate omitted text. The table manager is imported by the `USING` clause, making types `product_id` and `price` visible within the `pvc` module.

The complete specification of the pvc module can be found in Figure 1.7. The fact that pvc is a generic module and that `price_table_mngr` is used (and not macro expanded) is not depicted by the diagram in Figure 1.5.

```
pvc: MODULE [coin_detected, current_payment, product_availability_info,
             customer_selection, coin_return_status, sufficient_payment,
             change_due: TYPE]

  USING price_table_mngr          -- functional decomposition

  validate_payment: PROCESS
      [IN current_payment, IN coin_detected, IN coin_return_status,
       product_id, price
          -> OUT change_due, OUT sufficient_payment, sufficient_payment, product_id]
  get_valid_selection: PROCESS
      [IN product_availability_info, IN customer_selection, sufficient_payment
         -> OUT product_id, product_id]

  vp1  : ASSERT dflow(validate_payment, price_table_mngr!get_price, product_id)
  ptbl: ASSERT dflow(price_table_mngr!get_price, validate_payment, price)
  vp2  : ASSERT dflow(validate_payment, get_valid_selection, sufficient_payment)
  vs   : ASSERT dflow(get_valid_selection, validate_payment, product_id)

END pvc
```

Figure 1.7: Textual Specification of Product Vending Controller

In assertions `vp1` and `ptbl`, you will notice the name

```
price_table_mngr!get_price
```

This is a *fully qualified name*, that is, the name of a declared object prefixed by the name of the module in which the declaration appears and separated by an exclamation point. Unqualified names within a module must be unique. Consequently, every entity is identified uniquely by its fully qualified name. Qualification of names serves to disambiguate meanings when simple names are not sufficient. Hence, in our example, the qualification was unnecessary.

1.5 Vertical Hierarchies

A vertical hierarchy brings an existing horizontal hierarchy closer to an efficient implementation. Two levels in a vertical hierarchy are connected by an explicit mapping that describes how to interpret the concepts of the higher, more abstract diagram in terms of those of the lower, more concrete diagram. We will see how to specify this mapping in the next section. In this section, we construct three different kinds of refinements to the vending machine design:

- **Passive-object refinement.** Unstructured types will be given a suitable structure.
- **Active-object refinement.** Processes will be implemented in terms of sequential functions and shared data.
- **Dependency refinement.** The concept of data flow will be broken down into cases: signals, message-passing through sharing, and message-passing through copies.

The three kinds of vertical refinements will be illustrated by means of the product vending controller in Figure 1.5. In particular, we will focus on the "validate payment" process, including its external interface to the "get valid selection" process and its internal implementation.

1.5.1 Explicit Specification of Horizontal Levels

We start a vertical refinement by identifying the horizontal hierarchy that is the subject of the refinement. Therefore, we must state explicitly those modules that constitute each horizontal level in the vending machine design. (This information appears only textually in this document.)

The declarations

```
level1: LEVEL = vm_io, vending_machine, pvc, price_table_manager
level2: LEVEL = vm_io, vending_machine_impl, pvc_impl, price_table_manager_impl
```

specify two horizontal levels. Module **vm_io** does not change from one level to the next, but the other modules do change. The suffix "**_impl**" is intended to indicate an implementation module. However, this mnemonic device has no semantic significance.

```
level1: LEVEL = vm_io, vending_machine, pvc, price_table_manager
level2: LEVEL = vm_io, vending_machine_impl, pvc_impl, price_table_manager_impl
```

Figure 1.8: Textual Specification of Horizontal Levels

In general, the number of modules need not be the same at every level. One reason is that macro expansions may occur at lower levels, obviating the need to represent replaced objects. It is also possible that the basic module partitioning can vary from level to level.

The ordering of the levels in a vertical hierarchy as well as the relationships between levels are specified textually by means of a mapping specification. Later, we will use a mapping specification to say that `level2` is a vertical refinement of `level1` and to specify the relationship among the objects at the two levels. The ordering of modules within a horizontal level is given by the transitive closure of the `USING` and `REPLACE` clauses.

1.5.2 Passive-Object Refinement

We will specify the structure of the types that are in the external interface to process "validate payment" as well as the internal data structures used in its implementation. This will be done textually because a good visual representation of such definitions has not been developed.

The predefined or "built-in" types are `NUMBER` (the rational numbers), `INTEGER` (the integers), `NAT` (the positive integers), `BOOLEAN` (the values `true` and `false`), and `char` (characters that may or may not be printable). Constructed types are based, directly or indirectly, on the composition of built-in types.

The structured types in the external interface to `validate payment` are defined in Figure 1.9. Here are two examples from that figure.

```
coin_detected: TYPE = BOOLEAN
product_id   : TYPE = INTEGER [1..5]
```

The first declaration says that variables of type `coin_detected` may assume the value `true` or the value `false`. The second declaration assumes that we have that we have five different kinds of products. Therefore, type `product_id` has the integer value 1, 2, 3, 4, or 5.

We next implement the price table as an array of integers.

```
price_table: TYPE = ARRAY [1..8] OF INTEGER
```

Arrays are functions from the index type to the element type. In this example, `price_table` maps an integer in the range 1 to 8 into an integer.

```
vending_machine_impl: MODULE

  -- structure of some internal value domains

  coin_detected     : TYPE = BOOLEAN
  product_id        : TYPE = INTEGER [1..5]
  coin_return_status: TYPE = BOOLEAN

  payment           : TYPE = INTEGER  -- in cents
  change_due        : TYPE = INTEGER
  sufficient_payment: TYPE = BOOLEAN

END vending_machine_impl
```

Figure 1.9: Textual Implementation of Vending Machine Data Structures

```
product_price_table_impl: MODULE
  EXPORTING get_price
  get_price: PROCESS [product_id -> price]
                         -- representation of hidden price table
  price_table: TYPE = ARRAY [1..8] OF INTEGER
END product_price_table_impl
```

Figure 1.10: Textual Implementation of Price Table Data Structure

1.5.3 Active Object Refinement

We can implement the "validate payment" process in a number of ways. We will choose an implementation that consists of three sequential functions, a shared variable, and several dependencies that we have not seen in the previous development.

The "validate payment" process checks whether the amount paid is enough to pay for the selected product. If it is, "validate payment" sets output "sufficient payment" to true and issues correct change. Otherwise, "sufficient payment" is set to false. If "validate payment" receives a request to return the coins held by the machine, it returns them provided the product has not already been dispensed.

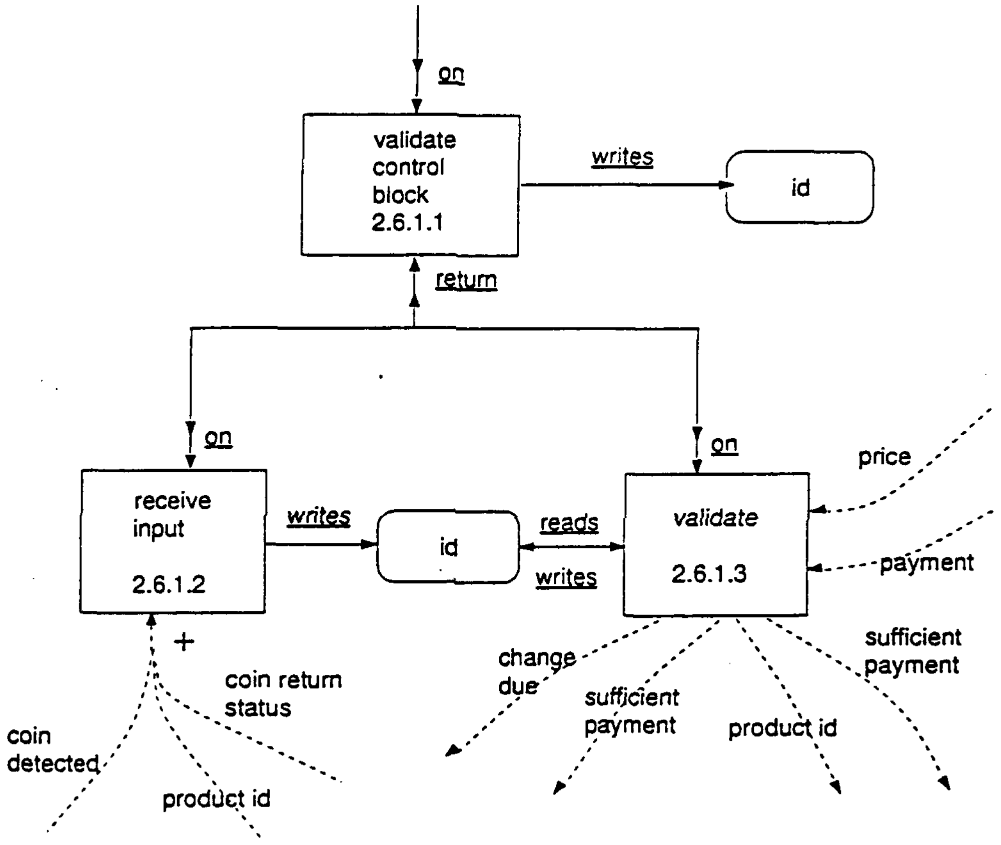

Figure 1.11: Implementation of Validate Payment Process

The design of the "validate payment" process is contained in Figure 1.11. The rectangles denote functions; the rectangle with rounded edges denotes a variable; and the underlined symbols on arrows are the names of relations. (The dataflow relation does not appear in this diagram.) Variable id is duplicated to avoid crossed lines in the figure. Duplication has no semantic consequences; that is, there is only one variable called id.

Process "validate payment" is activated by a function called "validate control block", which is indicated by the on signal at the top of the diagram. A signal is strictly a control relation; no data is transmitted. Process "validate control block" assigns a value to variable "id" and activates process "receive input". Process "receive input" waits for exactly one input; the "+" symbol on the arc denotes an n-ary exclusive-or relation. Process "receive input" writes variable "id" and returns control to "validate control block". The return of control is the same as for an ordinary subprogram call. Process "validate control block" activates process "validate" which waits for two inputs, and reads and writes the value of shared variable "id". Process "validate" produces three outputs.

Let us now turn to the textual representation of the diagram. The input to the "receive input" process is bundled into one type

```
vp_msg: TYPE = UNION(coin_detected, product_id, coin_return_status)
```

and the signature for "receive input" is

```
receive_input: FUNCTION [IN vp_msg -> product_id]
```

A UNION operator implicitly declares its arguments as *subtypes* of the defined type. The value domain of every subtype is assumed to be non-overlapping and a subset of the defined value domain. A record structure could have been used instead of a union type. However, that would require that callers of receive_input see the entire record structure even though only part of it is relevant to each caller.

The remaining objects are declared as follows.

```
validate: FUNCTION [IN payment
    -> OUT change_due, OUT sufficient_payment, OUT sufficient_payment]
validate_control_block: FUNCTION [ -> ]
id: VARIABLE product_id
```

The input to validate labeled "price" in the diagram is not a parameter of validate. It is a value returned as the result of a call (as yet unspecified) by validate to the price table manager. Function validate_control_block has no input or output; its sole purpose is to coordinate the other two functions. It writes variable id to synchronize with validate. Variable id will not be made visible to clients of "validate payment".

Figure 1.12 contains the implementation of "validate payment". The internal wiring in the diagram in Figure 1.11 is represented by relations pvc3–pvc10.

```
pvc_impl: MODULE [ALL]

  -- implementation of wiring

  pvc1: on_signal(get_valid_selection, validate_payment)
  pvc2: pass_message(get_valid_selection, validate_payment, product_id)

  -- implementation of validate_payment and its external interface

  validate_payment: PROCESS
    EXPORTING receive_input, validate, vp_msg

    -- external interface

    vp_msg: TYPE = UNION(coin_detected, product_id, coin_return_status)

    receive_input: FUNCTION [IN vp_msg -> product_id]
    validate: FUNCTION [IN price, IN payment
       -> OUT change_due, OUT sufficient_payment, OUT sufficient_payment]

    -- local objects

    validate_control_block: FUNCTION [ -> ]  -- internal activation block
    id: VARIABLE product_id                  -- local variable

    -- internal wiring

    pvc3: write(validate_control_block,id)
    pvc4: on_signal(validate_control_block, receive_input)
    pvc5: writes(receive_input,id)
    pvc6: return_signal(receive_input,validate_control_signal)
    pvc7: on_signal(validate_control_block, validate)
    pvc8: reads(validate, id)
    pvc9: writes(validate, id)
    pvc10:return_signal(validate, validate_control_block)

  END validate_payment

-- similar implementation for get_valid_selection goes here

END vending_machine_impl
```

Figure 1.12: Textual Implementation of Validate Payment Process and Its Wiring

1.5.4 Dependency Refinement

Having completed the internal design of process "validate payment", we are ready to consider its interface to process "get valid selection". Specifically, we will consider the manner in which the product id is transmitted.

In Figure 1.13 we can see that the dataflow arrow labeled "product id" has been split into two different kinds of arrows. The two arrows reflect the fact that data is transmitted in two steps. First, a wake-up signal is sent from "get valid selection" to "validate payment". Then, a value of type "product id" is sent as a message. More precisely, we have

```
on_signal(get_valid_selection, validate_payment)
pass_message(get_valid_selection, validate_payment, product_id)
```

labeled as `pvc1` and `pvc2` in the figure. This is an example of a dependency refinement. For it to be legal, PegaSys must prove that it is consistent with the meaning of the `dflow` relation.

1.6 Mapping Between Vertical Levels

A mapping describes how to interpret an abstract system description in terms of a more concrete one. In particular, we must place every object at the abstract level in one-to-one or in one-to-many correspondence with objects at the lower level. For example, in the context diagram in Figure 1.2, we must have a mapping for types `coin`, `slug`, `object`, `selection`, `coin_return_request`, `products`, and `product_status` as well as for processes `source` and `sink`. We do not need a mapping for process `vending_product` unless we want to duplicate it at the lower level. (Process `vending_product` was a replaceable process.)

In the specification of a mapping, associations can be omitted if the source and target names are the same. With respect to such a mapping, PegaSys must prove that the concrete level implements the more abstract level.

The intended associations for our example are shown in Figure 1.14. The `map1to2` module begins with a `MAPPING` clause that indicates that it is a mapping module that provides an interpretation of `level1` in terms of `level2`. Instead of describing objects and their dependencies, a mapping module lists associations such as

```
vs -> pvc1, pvc2
```

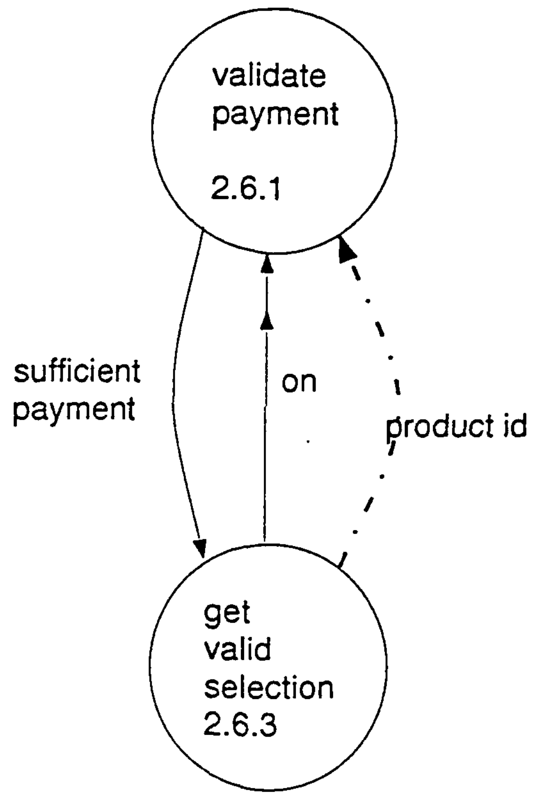

Figure 1.13: Implementation of `dflow` Relation

which says that the relation labeled **vs** in `level1` is to be interpreted by target relations `pvc1` and `pvc2` from `level2`. The other associations in the mapping module reflect differences in the two levels. The structured types at `level2`, except for `payment`, are not associated with types from `level1` because their names are the same at both levels.

```
map1to2: MODULE

MAPPING level1 ONTO level2

                  -> price_table                -- new object

  validate_payment -> validate_control_block,   -- process implementation
                       receive_input,
                       validate

  current_payment -> payment                    -- renaming

  validate_payment: PROCESS
     [IN current_payment, IN coin_detected, IN coin_return_status,
      product_id, price
      -> OUT change_due, OUT sufficient_payment, OUT sufficient_payment, product_id]
    ->
      validate_payment: PROCESS [IN vp_msg, IN payment, product_id, price
        ->  OUT change_due, OUT sufficient_payment, sufficient_payment, product_id]

  vs -> pvc1, pvc2                            -- dflow implementation

END mapping1to2
```

Figure 1.14: Mapping Module Connecting Vending-Machine Levels

1.7 Predefined and User-Defined Concepts

We have already discussed the primitive types: `NUMBER`, `INTEGER`, `NAT`, and `BOOLEAN`. Type `NAT` is a subtype of `INTEGER` and `INTEGER` is a subtype of `NUMBER`. The subtype relation is the transitive closure of the subtype declarations. A subtype can be coerced into the type of its parent type(s). We have types FUNCTION, MODULE, PROCESS, TYPE, and VARIABLE for typing basic objects.

The nine primitive relations are contained in Figure 1.15. The primitives can appear in specifications. They also can be used to define derived relations. The `dflow` relation used in our example was not a primitive. It is defined as follows.

```
dflow: FUNCTION [PROCESS, PROCESS, TYPE -> BOOLEAN]
dflow(x,y,z) = dataflow(x,y,z)
```

The signature says that `dflow` is applies to two processes and a type. The `dflow` relation is intended to be true provided the first process passes values of the specified type to the second process. The second equation defines `dflow` in terms of primitive `dataflow`. The `dataflow` relation is a more general form of data dependency. The `dflow2` relation is defined in the figure to apply only to sequential objects.

To see how these definitions work, consider the following simple example in which we implement a pure dataflow model by a sequential dataflow model. Let `L1` be the horizontal level defined by

```
i : integer
A : PROCESS [ -> INTEGER ]
B : PROCESS [ INTEGER -> ]
p : dflow(A, B, i)
```

To simplify the example, we omit the surrounding module. Let `L2` be the horizontal level defined by

```
j : integer
C : FUNCTION [ -> INTEGER ]
D : FUNCTION [ INTEGER -> ]
s : dflow2(C, D, i)
```

and let

```
i -> j
A -> C
B -> D
```

be a mapping `M` from `L1` onto `L2`. We can prove

$$M \vdash L2 \supset L1$$

using the definitions of `dflow` and `dflow2`, since both are defined in terms of the common base relation `dataflow`. Therefore, `L2` implements `L1` under the given mapping.

```
dependencies: MODULE
-- this module implicitly imported into all other modules

-- type definitions

  active_object: TYPE = MODULE | PROCESS | FUNCTION
  seqobject    : TYPE = MODULE | FUNCTION
  passive_object: TYPE = VARIABLE | TYPE | id: TYPE

-- primitives

  dataflow: FUNCTION [active_object, active_object, passive_object -> BOOLEAN]
  on_signal: FUNCTION [PROCESS, PROCESS --> BOOLEAN]
  off_signal: FUNCTION [PROCESS, PROCESS --> BOOLEAN]
  return_signal: FUNCTION [PROCESS, PROCESS --> BOOLEAN]
  pass_message: FUNCTION [PROCESS, PROCESS, TYPE -> BOOLEAN]
  writes: [active_object, VARIABLE --> BOOLEAN]
  reads:  [active_object, VARIABLE --> BOOLEAN]
  calls: [FUNCTION, FUNCTION, passive_object -> BOOLEAN]
  iflow: FUNCTION [active_object, active_object, passive_object -> BOOLEAN]

-- derived dependencies

  dflow: FUNCTION [PROCESS, PROCESS, TYPE -> BOOLEAN]
  dflow(x,y,z) = dataflow(x,y,z)

  dflow2: FUNCTION [seqobject, seqobject, passive_object -> BOOLEAN]
  dflow2(x,y,z) = dataflow(x,y,z)

END dependencies
```

Figure 1.15: Dependency Relations Used in the Example

It is important to note that the PegaSys methodology does not depend on a particular set of primitives or derived relations. The relations used for a particular development, however, must be specified in the standard module called `dependencies`. In general, it is probably best to encapsulate the primitive relations in a separate module, and then import that module into the `dependencies` module that defines the new relations tailored to the application.

Chapter 2

Tracking the Effects of Program Changes

2.1 Introduction

For large systems, it often is too difficult to predict the semantic effects of planned changes. The problem is inherently difficult, even for well-structured systems. But, in practice, it is nearly impossible because of "fine tuning" that tends to convolute the structural abstractions of the system.

Conventional formal methods offer little help. The question of whether a change to a program affects a certain system object boils down to determining whether a formula in the specification language is a theorem. This reduction would take place in a Hoare logic involving pre- and post-conditions, as well as in a logic based on the equivalence of functions. Unfortunately, the expressive behavioral specification languages are undecidable and some are incomplete. They also have insufficient mechanical theorem-proving support. Consequently, any approach based on a behavioral specification language would tend to be impractical for everyday use.

To obtain a practical solution, we make a sharp distinction between the kind of property to be analyzed and the kind of method used to analyze it. In particular, we reason about the semantic effects of changes through a structural analysis of a program. We believe that the right structural abstraction for capturing the "effects" relation between system objects is that of "information flow." Intuitively, information flows from an object x to an object y if, when the program is executed, a change in the value associated with x can change the value associated with y. This is a qualitative question in that we are only interested in whether *any* information flows from one object to another, not the amount of information that flows. For system objects x and y, a change to x is said to affect y provided the pair $\langle x, y \rangle$ is in the closure of the information-flow relation with respect to a set of special transitivity axioms. The axioms do not include the usual transitivity rule. If there is flow from x to y and from y to z, there is not necessarily flow from x to z.

We define a logic for approximating the direct and indirect information flows in a large program. Each construct in a programming language is described declaratively by rule of inference. Each rule is syntax-directed in that its application is driven by the abstract syntax of the programming language. The programming features covered include parameterized modules, procedures, global variables, functions without side effects, recursion, and various statements, such as assignment, while loop, and conditional. The entire logical system is concise and comprehensible.

Our formalization has three important characteristics that increase its practical utility. First, our logic is decidable, obviating the problems associated with semantic approaches. Decidability is achieved in part because we do not require formal, detailed specifications. Since programs are often constructed without any specification, this decision has the additional benefit of making our method more widely applicable. Second, *our logic is declarative and therefore new constructs can be handled simply by adding* more rules. Third, the implementation of our logic facilitates the interpretation of results. In particular, proofs are saved in a comprehensible form that makes explicit the justifications for each pair in a closure. Justifications are particularly useful in examining an approximation that is believed to be too inexact.

Because our logic is approximate and conservative, it has the logical property that it is complete but not sound. Let $\mathcal{I}$ denote the set of true information flows in a given program and let $\mathcal{A}$ denote our approximate inference system. In addition, let $x \Longrightarrow y$ indicate that there is information flow from object x to object y, where an object is a module, procedure, function, or variable. Then, we have

$$\text{if } \models_{\mathcal{I}} x \Longrightarrow y \text{ then } \vdash_{\mathcal{A}} x \Longrightarrow y$$

but the converse is false. Of course, the converse is desirable in classical logic, but, for our application, completeness is the crucial property. An overestimate (completeness and unsoundness) will not cause us to overlook an object affected by a change, but it may point to objects that are not relevant.

Another nice property of our axiomatization $\mathcal{A}$ is that failure to derive a flow means that the flow definitely does not occur. That is,

$$\text{if } \nvdash_{\mathcal{A}} x \Longrightarrow y \text{ then } \not\models_{\mathcal{I}} x \Longrightarrow y$$

which is just the contrapositive of the completeness property above.

The remainder of the chapter is organized as follows. The next section compares our work to related work involving the semantic and structural analysis of programs. Section 3 presents the abstract syntax for the language discussed in the body of the chapter. Section 4 gives a mathematical definition of information flow, illustrates its intransitivity, and defines rules for computing transitive flows across statements, including procedure calls. Section 5 introduces a logical method for referring to values of variables at specific program points. Section 6 shows how to state questions about changes and presents computer-generated analyses that answer positive and negative questions. The questions involve various program objects, including variables, procedures, and modules. Section 7 shows that changes are analyzed in polynomial time. Section 8 discusses modules and sketches how to handle a subtle example. Section 9 concludes with a brief summary of our results.

An earlier paper [24] presented similar results in a different logical framework. The main improvements are logical simplicity and uniformity, reduced execution costs, and the provision of meaningful justifications.

2.2 Related Work

2.2.1 Semantic Approaches

In 1972 Floyd [11] described an imagined interaction between a computer programmer and a formal program verification system that he believed might be feasible within the next decade. One of the main ideas in the scenario was for the computer to carry the burden of maintaining the consistency of specifications, programs, and lemmas following incremental changes. In 1978 Moriconi [23] developed and implemented a technique for this purpose based on a Hoare-style axiomatization of the programming language semantics. Most verification systems, past and present, are based at least implicitly on Hoare logic [18].

A proof of a program in Hoare logic is a sequence of steps, where each step is an instance of a Hoare axiom, a Hoare sentence derived from a previous step by a rule of inference, or a theorem in the underlying logic. Maintaining consistency in the presence of change boils down to determining theoremhood in the underlying theory (which is no easier than determining functional correctness). The underlying logic is determined by the specification language. The existing languages that we are familiar with are undecidable and, moreover, there typically is insufficient theorem proving power to handle the formulas that arise in practice. The undecidable specification languages include Anna [20, 21], EHDM [5], Gypsy [14], Larch [15, 16], OBJ [12, 13], VDM [3], and Z [17].

Perry [25, 26] recently suggested a similar approach based on Hoare logic for extending configuration management systems. He attempts to simplify matters by using the subset relation instead of logical implication to relate assertions. This transliteration works if the specifications are properly encoded in set theory. But the encoding offers no apparent gain, since the formulas to be proved are no simpler than before. Truth maintenance systems (e.g., [7, 10]) provide a different way of thinking about the problem, but we are still left with the intractable problem of testing for theoremhood.

2.2.2 Structural Approaches

Qualitative information flow has been studied extensively in the field of computer security by Denning [8] and others.[1] A program's security can be certified at compile-time through a conservative interpretation of the information-flow relation. A variety of formalisms have been used for this purpose, including attribute grammars [9] and logical rules [1]. Representative security analysis tools are those of McHugh and Good [22] and Rushby [29]. Work in computer security combines information flow considerations with security considerations. Moreover, the transitive information flows of interest here are not computed explicitly.

Bergeretti and Carré [2] use the concept of information flow in program development to detect certain kinds of errors and anomalies. Their work is more limited than ours in that it is oriented towards intraprocedural flows, although they do present preliminary ideas for procedures without recursion, without globals, and with very conservative assumptions about parameters. They adopt a relational approach for computing all possible facts, many of which may not be relevant to the specific change. Their relational approach does not address the problem of providing flow justifications. Our logical approach supports the derivation of specific results justified explicitly by formal proofs. We describe how to use the results of an analysis to reason about large-grain program objects, not just variables.

The information flow relation can be interpreted within a classical program flow-analysis framework. Only a crude interpretation can be provided using coarse-grain relations, such as the "calls" relation between procedures or the "uses" relation between modules. It appears that def/use chains could be put together across procedure boundaries to yield an interpretation equivalent to the one given in this chapter. (A

[1]Classical information theory, developed by Shannon [30] and others, is concerned with the *amount* of information generated by a particular event. We are interested in the simpler question of whether *any* information is generated by an event.

def/use chain represents the set of uses u of a variable x from a point p such that there is a path from p to u that does not redefine x.) Intraprocedural def/use chains have been used by Podgurski and Clarke [28] in defining a general notion of variable dependence that seems to be equivalent to intraprocedural information flow.

Our logical approach has a number of advantages over a graph-based flow-analysis framework that stem from differences in objectives. This report focuses on the abstract information-flow relation, the specification and prototyping of an analysis technique, and on the explication of analysis results. In contrast, program flow analysis has been studied primarily for use in optimizing compilers or other settings in which low-level relations and efficiency are of primary importance. In fact, our inference system can be viewed as a specification for a def/use implementation of the closure. Our logic can directly provide flow justifications, which would require a significant extension to a flow analysis implementation.

Recent work by Horwitz, Reps, and Binkley [19] is somewhat related. They describe a complex but efficient flow-analysis algorithm for computing program slices, a concept originally introduced by Weiser [31]. A slice is the set of all statements and predicates of a program that affect a variable at a given point. The computation of a slice inherently has a backward orientation, whereas tracking the effects of changes has a forward orientation. However, the assertions computed by our rules can be used to determine slices.

2.3 Abstract Syntax

We begin by focusing on programs that consist of a collection of (global) variables, functions, and procedures. Procedures can refer to global variables; functions always behave as pure mathematical functions. Parameters of procedures have a value-result semantics (copy-in/copy-out). Three kinds of statements are treated: assignment, a looping construct, and conditional.

Our logic does not depend on the concrete syntax of a particular programming language. Instead, it refers to an *abstract syntax* containing the features just described. The abstract syntax is defined in functional notation, specifically a many-sorted logic with subsorts. For example, the subsort declaration $Var \subseteq Expr$ means that every variable is an expression. Operators are defined in a mixfix syntax in which an underbar is a placeholder indicating where arguments should appear. This notation is borrowed from OBJ [13].

The abstract syntax for programs (without modules) is contained in Figure 2.1. To simplify the discussion, we assume that procedures, functions, and globals have unique names. In addition, locals of different procedures are distinct.

The discussion does not include structured objects and expressions with side effects, although we believe that our logic could be adapted to analyze them. Pointers and call-by-reference parameters can be added, but not as easily. Types are omitted from the abstract syntax because they are not used in the analysis.

2.4 Definition of Information Flow for Statements

2.4.1 Notation

We consistently use certain variables to range over particular classes of objects. The metavariable c ranges over constants of sort *Const*. Letters u, v, x, y, and z are metavariables ranging over variables and constants in the language. We let*primop* ranges over the primitive operators of the language (i.e., those that are not user defined), e, t_i $(i > 0)$, and b (for boolean) range over expression instances, and S and S_i range over statement instances. The letters f and p range over the names of functions and procedures, whose parameters are of kind k_i. Finally, the letter C denotes a context in which a particular analysis takes place. These naming conventions are summarized in Figure 2.2.

The predicates in Figure 2.3 will be used in defining information flow for the constructs in the abstract syntax. Two predicates are needed for statements, one for asserting flows across the statement and another for asserting which variables are modified directly or indirectly by the statement. Information can flow into an expression, so a predicate is needed to describe such flows. Interprocedural flow assertions model the variable bindings that result from a procedure call. The relation $\Longrightarrow_f$ denotes a flow from an actual to a formal and $\Longrightarrow_b$ denotes a formal to actual flow. The relations apply to implicit parameters, i.e., globals.

The context C is used in assertions to denote collected assumptions about the entities in a program. The term "context" as used here is equivalent to the term "environment" in denotational semantics. A context is a pair in which the first element is the set of global variables and the second is a mapping from procedure or function names to their descriptions. Specifically,

$$\text{Context} = \text{Globals} \times (\text{Name} \rightarrow \mathit{Kind} \times \mathit{ParamList} \times \mathit{Stmt})$$

where the sort *Globals* is a set of variables and *Kind* indicates whether the name is that

sorts

Const Expr ExprList Name Param
ParamKind ParamList PrimOp
Program Stmt Unit Var

subsorts

Var, Const $\subseteq$ Expr
PrimOp $\subseteq$ Name
Unit $\subseteq$ Program

operators

ExprList = List[Expr]

_(_) : Name ExprList $\rightarrow$ Expr

_ := _ : Var Expr $\rightarrow$ Stmt
_(_) : Name ExprList $\rightarrow$ Stmt
if _ then _ else _ fi : Expr Stmt Stmt $\rightarrow$ Stmt
while _ do _ od : Expr Stmt $\rightarrow$ Stmt
null : $\rightarrow$ Stmt
_;_ : Stmt Stmt $\rightarrow$ Stmt

value,value−result,result : $\rightarrow$ ParamKind
_ _ : ParamKind Var $\rightarrow$ Param
ParamList = List[Param]

var_ : Var $\rightarrow$ Unit
procedure _(_)_ : Name ParamList Stmt $\rightarrow$ Unit
function _(_)_ : Name ParamList Stmt $\rightarrow$ Unit
_ _ : Unit Program $\rightarrow$ Program

Figure 2.1: Abstract syntax (without modules).

Notation	Sort
c	Const
u, v, x, y, z	Var or Const
primop	PrimOp
e, t_i, b	Expr
S, S_i	Stmt
f, p	Name
k_i	ParamKind
$\mathcal{C}$	Context

Figure 2.2: Summary of Naming Conventions

of a procedure or function. The mapping also specifies the parameter list and body of the named entity.

Inference rules are used to axiomatize the basic information-flow predicates in Figure 2.3. Inference rules describe how assertions can be derived. An inference rule of the form

$$\frac{P_1 \ \ldots \ P_n}{C}$$

states that conclusion C can be inferred from the premises P_i. Each P_i and C is an instance of a predicate in Figure 2.3. If a rule has no premises, we write it without the horizontal bar. The rules are syntax-directed; at least one axiom or rule is given for each construct in the abstract syntax. The context referred to in an assertion can be derived from a program. We have implemented a program analyzer in Common Lisp that directly applies the rules given below to compute assertions.

The style of our inference rules is inspired by Plotkin's "structural operational semantics" [27]. This style of formalism is intended to produce concise, comprehensible definitions that are independent of internal representation details. The formalism has been used as a common framework for specifying, among other things, type checking, type inference, translation, and interpretation [4], and it is becoming a popular notation for language-directed specifications.

2.4.2 Mathematical Definition of Information Flow Predicates

The meaning of information flow can be illustrated with a few simple examples. Execution of the assignment statement `x:=y` causes flow from `y` to `x`. Execution of the conditional

Notation	**Interpretation** (with respect to context $\mathcal{C}$)
$\mathcal{C} \rhd [S]\, x \Longrightarrow y$	the value of x before execution of S affects the value of y after execution of S
$\mathcal{C} \rhd [S]\, \mathrm{mod}\; x$	execution of statement S may modify the value of variable x
$\mathcal{C} \rhd x \Longrightarrow \mathrm{val}(e)$	the value of x affects the value of expression e
$\mathcal{C} \rhd [S]\, x \Longrightarrow_f y$	intersubprogram forward flow from formal x to actual y for call S
$\mathcal{C} \rhd [S]\, x \Longrightarrow_b y$	interprocedural backward flow from actual x to formal y for call S
$\mathcal{C} \rhd \mathrm{global}(x)$	x is a global variable
$\mathcal{C} \rhd \mathrm{value}(p, i)$	the ith formal parameter of procedure p is a value parameter
$\mathcal{C} \rhd \mathrm{result}(p, i)$	the ith formal parameter of procedure p is a result parameter
$\mathcal{C} \rhd \mathrm{param}(p, i, x)$	the ith formal parameter of procedure p is x
$\mathcal{C} \rhd \mathrm{func}(p, S)$	p is a function with body S
$\mathcal{C} \rhd \mathrm{proc}(p, S)$	p is a procedure with body S

Figure 2.3: Summary of Predicates Used in Inference Rules

```
if x=0 then y:=0 else y:=1
```

causes a flow from x to y. A procedure call initiates a set of flows that reflect the actual/formal parameter bindings.

Before defining information flow, we introduce some sorts and functions. Let the sort *Val* denote the values of variables. The sort *Env* consists of mappings from variable names to values. The operations *val* and *set* retrieve and set the value, respectively, of a variable in an environment. The function *eval* evaluates an expression in a given environment and context; expressions have no side-effects. Function *exec* executes a statement in a given environment and context, and produces a new environment. Non-terminating execution produces the value "undefined." The signature for the operations is given below.

sorts Val Env**operators** val : Var Env → Val set : Var Val Env → Env eval : Expr Env Context → Val exec : Stmt Env Context → Env

We now make the following mathematical definitions:

$$\mathcal{C} \rhd [S]x \Longrightarrow y \text{ iff } \exists env\colon \mathrm{Env}, v\colon \mathrm{Val}[\mathrm{val}(y, \mathrm{exec}(S, env, \mathcal{C})) \neq \mathrm{val}(y, \mathrm{exec}(S, \mathrm{set}(x, v, env), \mathcal{C}))]$$

$$\mathcal{C} \rhd x \Longrightarrow \mathrm{val}(e) \text{ iff } \exists env\colon \mathrm{Env}, v\colon \mathrm{Val}[\mathrm{eval}(e, env, \mathcal{C}) \neq \mathrm{eval}(t, \mathrm{set}(x, v, env), \mathcal{C})]$$

$$C \triangleright [S] \bmod x \quad \text{iff} \quad \exists env\colon \mathrm{Env}[\mathrm{val}(x, env) \neq \mathrm{val}(x, \mathrm{exec}(S, env, C))]$$

These are the exact mathematical definitions of the first three predicates in Figure 2.3.

The first definition says that there is flow from x to y provided the value of y after execution of S differs when only the value of x is changed in the initial environment env. The second definition says that there is a flow from x to expression e if the value of e differs when only the value of x is changed. The third definition says that S modifies x provided the value of x after execution of S can be different from its value before.

Our inference rules approximate the mathematical definitions. We do not include the rules for defining the *mod* relation. They are straightforward and give a relatively exact interprocedural version of the "modifies" relation commonly used for program optimization [6].

The fact that the information flow relation is not transitive in the usual sense is illustrated by the example below.

Example 1 (*Intransitivity of information flow*) Consider the following program fragment:

```
procedure addinc(value-result sum, value-result i);
  add(sum,i); inc(i)

procedure add(value-result a, value-result b);
  a := a+b

procedure inc(value-result z);
  add(z,1)
```

Suppose that we want to know whether a change to the value of variable `sum` can affect the value of variable `z`. The call to `add` in `addinc` gives a flow from `sum` to `a` and the call from `inc` to `add` gives a backward flow from `a` to `z`. Hence, a flow from `sum` to `z` is in the transitive closure. But there is no execution sequence for which the value of `sum` affects `z`. The problem, of course, is that transitive flows are determined in part by the flow of control. For procedures, the interplay between control and information flow can be complex. □

2.4.3 Approximate Logic for Statements

Any constant or variable that appears in an expression affects the value of the expression.

expr-var

$$\mathcal{C} \rhd x \Longrightarrow \text{val}(x) \qquad x\colon \text{Var}$$

expr-const

$$\mathcal{C} \rhd c \Longrightarrow \text{val}(c)$$

expr

$$\frac{\mathcal{C} \rhd x \Longrightarrow \text{val}(t_i)}{\mathcal{C} \rhd x \Longrightarrow \text{val}(primop(t_1, \ldots, t_n))} \qquad i = 1, \ldots, n$$

The first rule says that any change in the value of x affects the value of x. The second one says that a constant affects its value. Strictly speaking, a constant cannot change, so there can be no information flow from a constant to something else. We include this rule because the programmer may edit a constant in a program, in which case we may want to see what depends on the constant. The third rule says that an change that affects any component of an expression affects the value of the expression. The sort *PrimOp* denotes built-in functions; user-defined functions are handled differently.

Anything that affects the value of the righthand side of an assignment affects the value of the variable on the lefthand side.

:=

$$\frac{\mathcal{C} \rhd x \Longrightarrow \text{val}(e)}{\mathcal{C} \rhd [y := e]\, x \Longrightarrow y}$$

It also is necessary to specify invariants over assignments. In particular, if an assignment does not modify some variable (i.e., the variable does not appear on the lefthand side), then the value of the variable before the assignment is said to affect its value afterwards.

not-mod

$$\frac{\neg(\mathcal{C} \rhd [S]\, \text{mod}\ x)}{\mathcal{C} \rhd [S]\, x \Longrightarrow x}$$

where, in practice, S can be restricted to be an assignment, the null statement, or a procedure call. Note that constants are always invariant across statements.

Statement composition is handled by the following rule:

seq

$$\frac{C \rhd [S_1]\, x \Longrightarrow y \quad C \rhd [S_2]\, y \Longrightarrow z}{C \rhd [S_1; S_2]\, x \Longrightarrow z}$$

Two flows are composed when the intermediate variable is the same and the two statements appear in sequence. In the absence of the not-mod rule, the composition rule could not be applied when one statement in a sequence does not modify a variable modified by another statement in the sequence.

Conditional statements are broken into two cases. The first deals with the flows on the two branches of the if-then-else. The second deals with the flow from the condition through the branches.

if

$$\frac{C \rhd [S_i]\, x \Longrightarrow y}{C \rhd [\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}]\, x \Longrightarrow y} \quad i = 1 \text{ or } i = 2$$

if-cond

$$\frac{C \rhd x \Longrightarrow \text{val}(b) \quad (C \rhd [S_1] \text{ mod } y \vee C \rhd [S_2] \text{ mod } y)}{C \rhd [\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}]\, x \Longrightarrow y}$$

The first rule says that the flows created by the statements in the branches are created by the conditional statement as a whole. The first premise of the second rule says that a variable x can affect the choice of the branch. The second premise says that variable y is affected by one of the branches. In this situation, x indirectly affects y. This rule does not take into account the fact that y could have the same value on both branches.

The while rules deal with three possibilities.

while-null

$$C \rhd [\text{while } b \text{ do } S \text{ od}]\, x \Longrightarrow x$$

while

$$\frac{C \rhd [\text{while } b \text{ do } S \text{ od}]\, x \Longrightarrow y \quad C \rhd [S]\, y \Longrightarrow z}{C \rhd [\text{while } b \text{ do } S \text{ od}]\, x \Longrightarrow z}$$

while-cond

$$\frac{C \rhd [\text{while } b \text{ do } S \text{ od}]\, x \Longrightarrow y \qquad C \rhd y \Longrightarrow \text{val}(b) \qquad C \rhd [S]\, \text{mod}\, z}{C \rhd [\text{while } b \text{ do } S \text{ od}]\, x \Longrightarrow z}$$

The first rule handles the situation in which the body S of the while loop is never executed. This means that the effect of the statement is exactly the same as the null statement. The second rule is recursive. If a flow from x to y is created by the while statement and a flow from y to z is created by the body S of the while statement, then the two flows can be composed. The third rule also is recursive, indicating that a transitive flow occurs when y affects condition C. The condition governs whether S is executed and therefore affects any variable modified by S.

We next deal with parameter passing in functions and procedures. Before stating the function and procedure rules, we first introduce rules for parameter passing. The first two rules deal with the transmission of values from a call site and the last two deal with return values. Globals and constants are implicit parameters at every call site. They are transmitted by the rule

$$\frac{C \rhd \text{global}(x) \;\vee\; x\!: \text{Const}}{C \rhd [p(t_1, \ldots, t_n)]\, x \Longrightarrow_f x}$$

which asserts that the value of a global or constant at the call site is the same as the value when the called procedure is entered. The rule

$$\frac{C \rhd \text{value}(p, i) \qquad C \rhd \text{param}(p, i, x) \qquad C \rhd u \Longrightarrow \text{val}(t_i)}{C \rhd [p(t_1, \ldots, t_n)]\, u \Longrightarrow_f x} \qquad i = 1, \ldots, n$$

asserts that a flow from a variable u into an actual parameter t_i is transmitted to the corresponding formal value parameter x_i. Globals are returned to themselves, analogously to the forward transmission of globals.

$$\frac{C \rhd \text{global}(x)}{C \rhd [p(t_1, \ldots, t_n)]\, x \Longrightarrow_b x}$$

Result parameters of procedures are transmitted back to the actual parameter.

$$\frac{C \rhd \text{result}(p, i) \qquad C \rhd \text{param}(p, i, x) \qquad t_i\!: \text{Var}}{C \rhd [p(t_1, \ldots, t_n)]\, x \Longrightarrow_b t_i} \qquad i = 1, \ldots, n$$

The first two premises assert that x is the i-th parameter of procedure p and x is also a result parameter. The third premise requires that t_i be a variable; from the conclusion, t_i must be an actual parameter in the call to p. Under these conditions, we can conclude that the call to p results in a backward flow from formal x to actual t_i.

We can now define the information-flow semantics of function and procedure calls. The rule for function calls is

func (expression)

$$\frac{C \rhd \text{func}(f, S) \quad C \rhd [f(t_1, \ldots, t_n)]\, u \Longrightarrow_f x \quad C \rhd [S]\, x \Longrightarrow \text{value}}{C \rhd u \Longrightarrow \text{val}(f(t_1, \ldots, t_n))}$$

The first premise checks that f is a function and S is its body. The second premise asserts that the call to f causes a forward flow from u to x; by the parameter passing rules, u and x are the same constant, the same global, or describe a flow from an actual to a formal. The last premise asserts that there is flow from x to the special program variable called *value*, which is used to indicate the return value of a function. The conclusion says that the value of u affects the value of the call.

The procedure call rule is complicated by the possibility of multiple backward flows. The idea behind the rule is that a forward flow into a procedure can be passed through the procedure through transitive local flows and then back to the caller via a backwards flow.

proc (statement)

$$\frac{C \rhd \text{proc}(p, S) \quad C \rhd [p(t_1, \ldots, t_n)]\, u \Longrightarrow_f x \quad C \rhd [S]\, x \Longrightarrow y \quad C \rhd [p(t_1, \ldots, t_n)]\, y \Longrightarrow_b v}{C \rhd [p(t_1, \ldots, t_n)]\, u \Longrightarrow v}$$

The first premise asserts that p is a procedure and S is its body. The second premise asserts that the call to p results in a forward flow from u to x. The third says that there is a local flow from x to y. The last requires a backward return flow from y to v. From these four conditions, we can infer that the call to p has the net effect of causing a flow from u to v.

2.5 Variables at Program Points

The assertions in Figure 2.3 involve variables that denote values before and after a given program statement. They do not allow us to make assertions that relate variables at two arbitrary points in a program. In addition, we cannot ask whether a change to a local variable of a given procedure can affect the value of a local of another procedure, since the locals are in different scopes. To provide this capability, we provide a mechanism for introducing names, which have global scope, for the values of variables at specific points in a program. The new names are called *label variables* of sort *LabelVar* (a subsort of *Var*). For the purposes of this report, a variable ending in "0" is a label variable, otherwise it is an ordinary variable. One way to introduce label variables involves modifying the program; another requires no modification but involves new inference rules. Both approaches are presented below.

For a given variable and point, we may be interested in tracing flows forward, backward, or both. To trace forward from a point between two statements, we insert the assignment `x:=exp(x,x0)` where `x` is the variable of interest and `x0` is a new unique global. Primitive operator `exp` has the property that its value depends on `x` and `x0`. This follows from a direct application of the expression rule (**expr**). To trace backward, we insert `x0:=exp(x,x0)` and both are needed to trace both directions. An example is given in the next section.

Although this approach is simple, it is unattractive in the sense that we must modify the program. This is particularly serious if we are interested in a large number of program points. Fortunately, modification of a program is not necessary, as we can introduce label variables during the inference process. For this purpose, we introduce the following rules.

$$\frac{C \rhd [S]\, x \Longrightarrow y}{C \rhd [S]\, l \Longrightarrow y}$$

$$\frac{C \rhd [S]\, x \Longrightarrow y}{C \rhd [S]\, x \Longrightarrow l}$$

$$C \rhd l \Longrightarrow \text{val}(e)$$

where l is a label variable. It is necessary to record the association among label variable, the renamed ordinary variable, and the statement or expression in order to properly interpret results of an analysis. For example, in the first rule, l represents the value of x before execution of statement S. Renamings must be complete and uniform for this approach to be equivalent to the previous one that introduced assignments.

Label variables have two important properties. The first is that they are treated as globals by the parameter passing rules, allowing them to be moved from scope to scope.

Notation	Interpretation (with respect to context C)
$C \rhd \text{orig}(l, x)$	x is the variable associated with label variable l
$C \rhd \text{varof}(x, p)$	x is a variable referenced in procedure or function p
$C \rhd \text{sub}(p, S)$	p is a procedure or function with body S
$C \rhd \text{subof}(p, M)$	procedure or function p is in module M

Figure 2.4: Summary of Predicates Used in Questions

Second, there is always a flow from a label variable to itself across all statements. That is,

$$C \rhd [S]\, l \Longrightarrow l$$

This is guaranteed in the first approach by the choice of assignments and in the second because no assignments to l can exist. This fact is used to propagate labeled flows through statement sequences.

2.6 Deducing the Effects of Program Changes

We want to ask questions about changes to a number of different kinds of objects: variables (including globals), procedures, functions, and parameterless modules. Questions involving large-grain objects are reduced to questions involving only our assertions about statements, possibly involving label variables. For example, a change to variable v affects module M provided v flows into a variable associated with M. In general, the questions of interest have the following pattern: Does a change to object X affect object Y?

A query can be any first-order formula with finite quantification. This means that we can quantify over the objects in a program, such as its modules or procedures. An analysis of the program (using the inference rules of the previous sections) produces all of the ground (variable-free) facts about the program. These facts are positive and facts not in this set are assumed to be false. First-order queries are defined recursively in terms of the ground facts. For a specific program, sorts are interpreted with respect to the objects in the current program. For example, $x\colon Var$ indicates that x ranges over the finite set of variables in the current program, not the countably infinite set of variables that could occur in a program. Formulas in this section will make use of four new relations, which are summarized in Figure 2.4.

We will find it convenient to have notation for asserting that execution of a procedure or function creates a certain flow. For a name P, we have

$$C \rhd [P]\, x \Longrightarrow y \text{ iff } (\exists S)[C \rhd \mathrm{sub}(P,S) \wedge C \rhd [S]\, x \Longrightarrow y]$$

This says that there is a flow from x to y for P if and only if P is a procedure or function subprogram in context C and there is a flow from x to y in its body S.

Example 2 (*Absence of an interprocedural flow*) Consider the following program

```
procedure addinc(value-result sum, value-result i);
  add(sum,i); inc(i)

procedure add(value-result a, value-result b);
  a := a+b

procedure inc(value-result z);
  add(z,1)
```

Our implementation of the inference rules produces the following assertions for the body of addinc.

```
0: [add(sum,i); inc(i)]i=>sum
1: [add(sum,i); inc(i)]sum=>sum
2: [add(sum,i); inc(i)]i=>i
3: [add(sum,i); inc(i)]1=>i
4: [add(sum,i); inc(i)]1=>1
```

Suppose that we are interested in whether the value of sum on entry to addinc affects the value of i on exit. Formally, we want to know whether $C \rhd [addinc]\, sum \Longrightarrow i$ and it is easy to see that it is false. Note that there is no need for label variables in this example, since the basic assertion deals with before and after values for addinc. Because approximations are conservative, we know that there really is no flow from sum to i. □

Example 3 (*Presence of an interprecedural flow*) Suppose that we are interested in whether the value of i before the call to inc affects the value of a on entry to add. To answer this question, we introduce label variables i0 and a0.

```
var i0, a0;

procedure addinc(value-result sum, value-result i);
  add(sum,i); i := exp(i,i0); inc(i)

procedure add(value-result a, value-result b);
  a0 := exp(a,a0); a := a+b

procedure inc(value-result z);
  add(z,1)
```

The new assignment in addinc associates i0 with the value of i before the call to inc. The one in add associates the value of a upon entry with a0. The assignments have a different form because i0 is to be propagated forward and a0 backward.

We want to find a procedure P in our program such that $C \triangleright [P]\, i0 \Longrightarrow a0$. Of the ground facts generated by the computer, here are the ones for addinc.

```
0: [...]i0=>i0
1: [...]i0=>a0
2: [...]i0=>i
3: [...]i=>a0
4: [...]i=>i
5: [...]a0=>a0
6: [...]sum=>a0
7: [...]sum=>sum
8: [...]i=>sum
9: [...]1=>i
10: [...]1=>1
```

Ellipses denote the body of addinc.

We can see that the second assertion validates the desired flow, i.e., it proves $C \triangleright [addinc]\, i0 \Longrightarrow a0$. Since this is a positive assertion, there is no guarantee that the flow actually occurs. Below is a formal machine-generated proof that validates this assertion.

```
Proof of [add(sum,i); i := exp(i,i0); inc(i)]i0=>a0

(1)  [add(sum,i)]i0=>i0                    - not-mod i0
          Also proc[add(sum,i)]i0 =>f i0 [a0 := exp(a,a0); a := a+b]i0=>i0
```

```
                  =>b i0
(2)  i0=>[i0]                                - expr-var
(3)  i0=>[exp(i,i0)]                         - expr[2] (2)
(4)  [i := exp(i,i0)]i0=>i                   - := (3)
(5)  [add(sum,i); i := exp(i,i0)]i0=>i
                                             - seq (1) (4)
(6)  [inc(i)]i =>f z                         - =>f[1]
(7)  [add(z,1)]z =>f a                       - =>f[1]
(8)  a=>[a]                                  - expr-var
(9)  a=>[exp(a,a0)]                          - expr[1] (8)
(10) [a0 := exp(a,a0)]a=>a0                  - := (9)
(11) [a := a+b]a0=>a0                        - not-mod a0
(12) [a0 := exp(a,a0); a := a+b]a=>a0        - seq (10) (11)
(13) [add(z,1)]z=>a0                         - proc[1->] (7) z =>f a (12) a0 =>b a0
(14) [inc(i)]i=>a0                           - proc[1->] (6) i =>f z (13) a0 =>b a0
(15) [add(sum,i); i := exp(i,i0); inc(i)]i0=>a0
                                             - seq (5) (14)
```

The justifications are keyed to the labels on the rules. The proof shows how the flow from i0 to a0 actually occurs, including the relevant control path. Steps (1)–(5) establish that i0, starting at the new assignment in addinc, flows into the value of i immediately before the call to inc. Steps (8)–(12) verify that there is a flow from the value of a on entry to add to the point associated with a0. Steps (7) and (13) are assertions about the body of inc, verifying an interprocedural flow from formal z of inc to a0. Steps (6) and (14) verify that the call to inc creates a flow from i to a0. The last step composes the assertions at (5) and (14) creating the desired flow for the body of addinc. □

We now consider more general questions. In the formulas below, free variables in formulas can be instantiated to form a specific question. For simplicity, we assume that label variables have been introduced for every variable at every program point. (In practice, the number of label variables can be reduced based on the particular question.)

Example 4 (*Effect on a variable*) Suppose that we are interested in whether a change to a variable x affects a variable y. The formula

$$(\exists P\colon \mathrm{Name})(\exists u, v\colon \mathrm{LabelVar})[\mathcal{C} \rhd \mathrm{orig}(u, x)) \wedge \mathcal{C} \rhd \mathrm{orig}(v, y) \wedge \mathcal{C} \rhd [P]\, u \Longrightarrow v],$$

where x and y are free, says that a change to a variable x can affect the value of a variable y if a change to a label variable u associated with x can affect a label variable v associated with y when some procedure P is executed.

Our earlier question about whether *sum* affects *i* can be stated as an instance of this formula. Substituting *sum* for x and i for y, we obtain

$$(\exists P\colon \text{Name})(\exists u, v\colon \text{LabelVar})[\mathcal{C} \rhd \text{orig}(u, sum) \wedge \mathcal{C} \rhd \text{orig}(v, i) \wedge \mathcal{C} \rhd [P]\, u \Longrightarrow v],$$

We did not use label variables before, but this formulation is equivalent. □

Example 5 (*Effect on a procedure*) To ask whether a change to a variable x affects an arbitrary procedure P, we use the defining formula

$$(\exists R\colon \text{Name})(\exists u, v\colon \text{LabelVar})(\exists y\colon \text{Var})[\mathcal{C} \rhd \text{orig}(u, x) \wedge \mathcal{C} \rhd \text{orig}(v, y) \wedge \mathcal{C} \rhd \text{varof}(y, P) \wedge \mathcal{C} \rhd [R] u \Longrightarrow v],$$

where x and P are free. Observe that R can be any procedure, including P. It will be different from P when the procedure that owns x is not called, directly or indirectly, by P.

If instead we are interested in whether the value of x at a certain point affects a procedure P, we would would use the formula

$$(\exists R\colon \text{Name})(\exists v\colon \text{LabelVar})(\exists y\colon \text{Var})[\mathcal{C} \rhd \text{orig}(v, y) \wedge \mathcal{C} \rhd \text{varof}(y, P) \wedge \mathcal{C} \rhd [R]\, u \Longrightarrow v],$$

where u is free and to be instantiated with the label variable for x at the point of interest. □

Example 6 (*Effect on a module*) A change to a variable x can affect module M if the change affects a procedure contained in M. That is, we must prove an instance of

$$\begin{aligned}(\exists P, R\colon \text{Proc})(\exists u, v\colon \text{LabelVar})(\exists y\colon \text{Var})[\mathcal{C} \rhd \text{orig}(u, x) \wedge \mathcal{C} \rhd \text{orig}(v, y) \wedge \\ \mathcal{C} \rhd \text{varof}(y, P) \wedge \mathcal{C} \rhd \text{subof}(P, M) \wedge \mathcal{C} \rhd [R]\, u \Longrightarrow v].\end{aligned}$$

where x and M are free. □

2.7 Complexity

The time complexity of our inference algorithm is linear in the size of the program and polynomial with respect to the total number of variables and constants. For a large program, the size of the program usually should dominate.

In abstract syntax trees, different copies of the same syntactic structure are treated as distinct. The parameters used in the following analysis are given below.

c	number of constants in program
g	number of global variables in program
l	number of locals in a procedure (over all instances)
v	number of vars $(g + l)$
s	program size (number of nodes in tree)

Label variables are counted as globals.

The basic evaluation strategy involves an initial pass to compute invariant or static parameter passing relations, the *mod* relation, and an initial assertions, followed by the application of a worklist-based inference algorithm. Most of the rules for the parameter passing relations can be applied in an initial pass of the program, since they are invariant over the inference process. The cost of this is small in comparison to total cost, so the details are omitted.

The inference process is carried out by a worklist algorithm. The elements of the worklist are assertions of the form $C \triangleright [S]\, x \Longrightarrow y$, $C \triangleright x \Longrightarrow \mathrm{val}(e)$, or $C \triangleright [p(t_1, \ldots, t_n)]\, u \Longrightarrow_f x$. The worklist is initialized by a first scan of the program that applies the direct rules requiring no antecedent conditions (such as **expr-var**, **expr-const**, **:=**, and **not-mod**). The worklist of new assertions is processed until it is empty. When an assertion is removed from the worklist, all possible derived assertions are created and the new ones are added to the worklist.

The total cost of applying the inference rules is bounded at a given node by the cost of systematically applying the rules for all possible subsidiary assertions. The bound on the total number of assertions for any program element is $(c + v)v + c$. The worklist algorithm propagates new assertions in a complex pattern, but the total cost paid is just the sum of the incremental costs of exploring the possible new consequences of each subsidiary assertion at each node. For example, in the **seq** rule, if a new assertion $C \triangleright [S_1]\, x \Longrightarrow y$ is considered, we need to find all assertions $C \triangleright [S_2]\, y \Longrightarrow z$ that might be used with this assertion in the rule. There can be at most v such assertions and so the incremental cost is v. There are $(c + v)v + c$ possible assertions so the total cost is roughly $(c + v)v^2$ (ignoring some special cases associated with constants). The analysis is the same for an assertion coming in on the right, since the cost is always the total number of possible antecedents of the rule.

The **while** rule is costly since the incremental cost of an assertion is the cost of doing a simple transitive closure process. (This probably could be improved with a more sophisticated algorithm.) The cost of applying inference rules at a **while** node is $(c + v)v^3 + cv^2 + c$.

The total cost of information flow analysis is the sum of the costs of all the program elements:

$$O(s(c+v)v^3)$$

If there are no while loops or no recursive procedures, the cost would be

$$O(s(c+v)v^2)$$

We have assumed that the cost of adjoining a variable to a set of variables is constant. In practice, the cost may depend on implementation details. The actual cost may be $c+v$, which would be an additional factor in the above cost formulas.

2.8 Extensions

2.8.1 Parameterized Modules

A module consists of variables, functions, and procedures. A parameter to a module can be a variable, function, or procedure. Functions and procedures that are passed as values cannot reference global variables.

The basic idea is to use assumptions about the parameters of a module to derive conditional results (summary information) that depend on those assumptions. For a particular instantiation of the parameterized module, we can discharge assumptions to get specific unconditional results. When doing analysis under assumptions A, the existing rules are used along with some special rules that involve conditions in A. If an assertion P is a result of this analysis, then the conditional summary is $A \supset P$. We take this approach for simplicity; it would be better to associate assumptions with individual assertions.

In the analysis of variable parameters, we must know which formals correspond to the same actuals. The assertion $x \equiv y$ says that formals x and y are instantiated with the same actual variable. The assumptions for variables are a conjunction of assertions of this form.

The special rules say that equivalent variable parameters can be interchanged in assertions. One such rule is

$$\frac{C \rhd [S]\, u \Longrightarrow x}{C \rhd [S]\, u \Longrightarrow y} \quad \text{if } x \equiv y$$

The assumptions for procedures are of the form

$$[p(x_1,\ldots,x_n)]\, x_i \Longrightarrow x_j$$

$$[p(x_1,\ldots,x_n)]\, c \Longrightarrow x_j$$

where the x_i are considered to be specific variables and c is any constant. The first assertion says that a call to procedure p creates a flow from the ith parameter to the jth parameter. The second assertion creates a flow from a constant. An example of a special rule for procedures is

$$\frac{C \rhd u \Longrightarrow \mathrm{val}(t_i) \quad t_j\!: Var}{C \rhd [p(t_1,\ldots,t_n)]\, u \Longrightarrow t_j} \quad \text{if } C \rhd [p(x_1,\ldots,x_n)]\, x_i \Longrightarrow x_j$$

The rules for functions are similar.

There can be a problem with combinatorial explosion since arbitrary subsets of the conditions on the parameters may appear as conditions in the results of analysis of the parameterized object. In practice, it may be preferable to wait until the actual parameters are given before attempting an analysis.

2.8.2 A Difficult Example

Weiser's paper on slicing [31] presents an example which shows the limitations of the method presented in that paper. The fundamental problem in the example appears not to have been addressed in the literature. The same problem can occur when reasoning about information flows.

Here is Weiser's example:

```
A := constant
WHILE P(k) DO
   IF Q(C) THEN BEGIN
     B := A
     X := 1
   ELSE BEGIN
     C := B
     Y := 2
     END
```

```
      K := K + 1
      END
   Z := X + Y
   WRITE(Z)
```

Our analysis technique would indicate incorrectly that there is a flow from `constant` to `Z`. However, any execution path where the value of `A` has affected the value of `C`, in which case the value of `A` might indirectly affect the value of `X` or `Y` (and hence `Z`), both `X` and `Y` have already been assigned constant values that are not changed by either branch of the conditional. Therefore, no conditional flow from `A` to `Z` can occur.

To correctly analyze this program, it is necessary to keep track of the information flows that occur *together* along the same path and to require, for conditional flows, that there be a different modification of the dependent variable in the two branches.

Let the new assertion

$$\mathcal{C} \rhd [S]\, x \rightarrow y$$

have logical definition

$$\mathcal{C} \rhd [S]\, x \rightarrow y \;\text{ iff }\; \forall c{:}\,\mathrm{Env}[\mathrm{val}(x,c) = \mathrm{val}(y, \mathrm{exec}(S,c,\mathcal{C}))]$$

which asserts that execution of S has the logical effect of the assignment "y := x". We treat $x \rightarrow y$ as a separate syntactic entity that can occur in more complex expressions.

The special connectives $\wedge$ and $\vee_U$ (where U is, in general, a set of variables) have similar properties to the familiar logical connectives, having commutative and associative laws (the details are tricky for $\vee_U$), and so forth. The general form of a statement assertion is

$$\mathcal{C} \rhd [S]\, A$$

where A is formed using $x \rightarrow y$ assertions and the $\wedge$ and $\vee_U$ connectives. During analysis, a single assertion of this form is derived for each statement. An analysis successively refines the assertion until a fixpoint is reached.

An expression of the form $A \vee_U B$ corresponds to a logical expression of the form $(C(U) \wedge A) \;\vee\; (\neg C(U) \wedge B)$, where $C(U)$ is the predicate of a conditional expression.

The flow assertion $x \rightarrow y$ indicates an explicit unconditional flow. Given an assertion $C \rhd [S]\, A$, there is a conditional dependence on a variable x if $\vee_U$ occurs in A and $x \in U$. The variables modified (i.e. occurring on the right of a $\rightarrow$) in the arguments to the occurrence of $\vee_U$ are conditionally dependent on x.

The following rules are used to analyze the program.

$$\frac{C \rhd [S]\, A \qquad C \rhd [S]\, B}{C \rhd [S]\, A \wedge B}$$

$$\frac{\neg C \rhd [S] \operatorname{mod} x}{C \rhd [S]\, x \rightarrow x}$$

$$C \rhd [y := x]\, x \rightarrow y$$

$$\frac{v \neq y}{C \rhd [y := x]\, v \rightarrow v}$$

$$\frac{C \rhd [S_1]\, A \qquad C \rhd [S_2]\, B}{C \rhd [S_1; S_2]\, (A; B)}$$

$$\frac{C \rhd U \Longrightarrow \mathrm{val}(b) \qquad C \rhd [S_1]\, A \qquad C \rhd [S_2]\, B}{C \rhd [\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}]\, (A \vee_U B)}$$

In the last rule, U may be a set of variables. The last occurrence of ";" in the seq rule is a new operator satisfying the distributive laws

$$(A \vee_U B); C = (A; C) \vee_U (B; C)$$

$$C; (A \vee_U B) = (C; A) \vee_{(C;U)} (C; B)$$

where $C; U$ is the inverse image of the set of variables in U under the basic flow mappings in C. (It can be arranged that C is a conjunction of these by always applying the first distributive law first). If A and B are conjunctions of basic $\rightarrow$ assertions, then $A; B$

is a conjunction of basic $\rightarrow$ assertions consisting of the assertions obtained by chaining assertions from A with assertions from B. That is, if $x \rightarrow y$ is in A and $y \rightarrow z$ is in B, then $A; B$ includes $x \rightarrow z$.

A critical property of $\vee_U$ is the following idempotence law

$$A \vee_U A = A$$

This captures the idea that if there is no difference in the two branches or cases of a conditional, then there is really no conditionality.

A simple example of the problem in Weiser's program is illustrated by the following program fragment.

```
if q(c) then b := a; x := 1 else c := b; y := 2 fi;
if q(c) then b := a; x := 1 else c := b; y := 2 fi;
if q(c) then b := a; x := 1 else c := b; y := 2 fi;
```

We are interested in whether there is a conditional flow from **a** to **x** or **y**. If we analyze this program fragment, we derive several assertions, including

$$\mathcal{C} \rhd [\mathrm{b := a; x := 1}]\,(\mathrm{a \rightarrow a \wedge a \rightarrow b \wedge c \rightarrow c \wedge 1 \rightarrow x \wedge y \rightarrow y})$$
$$\mathcal{C} \rhd [\mathrm{c := b; y := 2}]\,(\mathrm{a \rightarrow a \wedge b \rightarrow b \wedge b \rightarrow c \wedge x \rightarrow x \wedge 2 \rightarrow y})$$
$$\mathcal{C} \rhd [\text{if q(c) then b := a; x := 1 else c := b; y := 2 fi}]$$

$$(\mathrm{a \rightarrow a \wedge a \rightarrow b \wedge c \rightarrow c \wedge 1 \rightarrow x \wedge y \rightarrow y})$$

$$\vee_{\{c\}}(a \rightarrow a \wedge b \rightarrow b \wedge b \rightarrow c \wedge x \rightarrow x \wedge 2 \rightarrow y)$$

These assertions precisely describe the effects of parts of the program fragment.

Analysis of one if statement gives

$$(a \rightarrow a \wedge a \rightarrow b \wedge c \rightarrow c \wedge 1 \rightarrow x \wedge y \rightarrow y)$$

$$\vee_{\{c\}}(a \rightarrow a \wedge b \rightarrow b \wedge b \rightarrow c \wedge x \rightarrow x \wedge 2 \rightarrow y)$$

Let A denote this expression. Then, the result of the analysis of the complete program fragment is $A; A; A$. Let C be the assertion $(a \rightarrow a \wedge a \rightarrow b \wedge a \rightarrow c \wedge 1 \rightarrow x \wedge 2 \rightarrow y)$.

The key point in simplifying $A; A; A$ is that the only context in which the variable a occurs in a $\vee_U$ is $C \vee_{\{a\}} C$, eliminating the conditional dependence on a.

This completes the sketch of a logical method for handling the fundamental problem in Weiser's example. It is not at all clear how this can be done in a graph-based flow analysis framework. Graph-based methods treat individual dependencies in isolation and don't extend naturally to situations in which combinations of flows must be considered.

2.9 Conclusion

Reasoning about changes is necessary in practical software development primarily due to continual changes in requirements and the support environment. We have developed and implemented a logical technique for determining the semantic effects of program changes based on an analysis of the abstract syntax of a generic programming language containing many of the features used in building large systems. A new idea behind the logic is that of approximate reasoning about changes based on a conservative interpretation of the semantic information-flow relation. Our logical formalization has several advantages over competing formalizations and is comparable in efficiency to the best alternative formalization in a program flow-analysis framework. We hope that automatic formal reasoning about the direct and indirect effects of changes will become a standard component of everyday programming environments.

Bibliography

[1] G.R. Andrews and R.P. Reitman. An axiomatic approach to information flow in parallel programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, January 1980.

[2] J.-F. Bergeretti and B.A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[3] D. Bjørner and C.B. Jones. *Formal Specification and Software Development.* Series in Computer Science. Prentice-Hall International, 1982.

[4] D. Clément, J. Despeyroux, T. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. In *Proceedings of the France-Japan Artificial Intelligence and Computer Science Symposium*, pages 49–89, October 1986.

[5] *The* EHDM *Specification Language.* Computer Science Laboratory, SRI International, May 1989.

[6] K.D. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 247–258, June 1984.

[7] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[8] D.E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.

[9] D.E. Denning and P.J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[10] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[11] R.W. Floyd. Toward interactive design of correct programs. *Proceedings of IFIP Congress 71*, pages 7–10, 1972.

[12] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings of the Twelfth Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[13] Joseph Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Laboratory, August 1988.

[14] D.I. Good, R.L. Akers, and L.M. Smith. Report on Gypsy 2.05. Technical Report CLI–1, Computational Logic Inc., Austin, Texas, 1986.

[15] J.V. Guttag, J.J. Horning, and J.M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[16] J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in five easy pieces. Technical Report 5, Digital Systems Research Center, Palo Alto, California, July 1985.

[17] I. Hayes, editor. *Specification Case Studies.* Series in Computer Science. Prentice-Hall International, 1987.

[18] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[19] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 35–46, Atlanta, Georgia, June 1988.

[20] D.C. Luckham and F.W. von Henke. An overview of Anna, A specification language for Ada. *IEEE Software*, 2(2):9–23, March 1985.

[21] D.C. Luckham, F.W. von Henke, B. Krieg-Brückner, and O. Owe. *ANNA, A Language for Annotating Ada Programs*, volume 260. Springer-Verlag Lecture Notes in Computer Science, 1987.

[22] J. McHugh and D.I. Good. An information flow tool for Gypsy. In *Proc. 1985 Symposium on Security and Privacy*, pages 46–48, Oakland, California, April 1985. IEEE Computer Society.

[23] M. Moriconi. A designer/verifier's assistant. *IEEE Transactions on Software Engineering*, SE-5(4):387–401, July 1979. Reprinted in *Artificial Intelligence and Software Engineering*, edited by C. Rich and R. Waters, Morgan Kaufmann Publishers, Inc., 1986. Also reprinted in *Tutorial on Software Maintenance*, edited by G. Parikh and N. Zvegintzov, IEEE Computer Society Press, 1983.

[24] M. Moriconi. A practical approach to semantic configuration management. In *Proceedings of ACM SIGSOFT Conference on Software Testing, Analysis, and Verification*, pages 103–113, Key West, Florida, December 1989.

[25] D.E. Perry. Software interconnection models. In *Proceedings of the 9th International Conference on Software Engineering*, pages 61–69, Monterey, California, March 1987.

[26] D.E. Perry. Version control in the Inscape environment. In *Proceedings of the 9th International Conference on Software Engineering*, pages 142–149, Monterey, California, March 1987.

[27] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[28] A. Podgurski and L.A. Clarke. The implications of program dependencies for software testing, debugging, and maintenance. In *Proceedings of ACM SIGSOFT Conference on Software Testing, Analysis, and Verification*, pages 168–178, Key West, Florida, December 1989.

[29] J.M. Rushby. The security model of Enhanced HDM. In *Proceedings 7th DoD/NBS Computer Security Initiative Conference*, pages 120–136, Gaithersburg, Maryland., September 1984.

[30] C.E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423 (July), 623–656 (October), 1948.

[31] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.