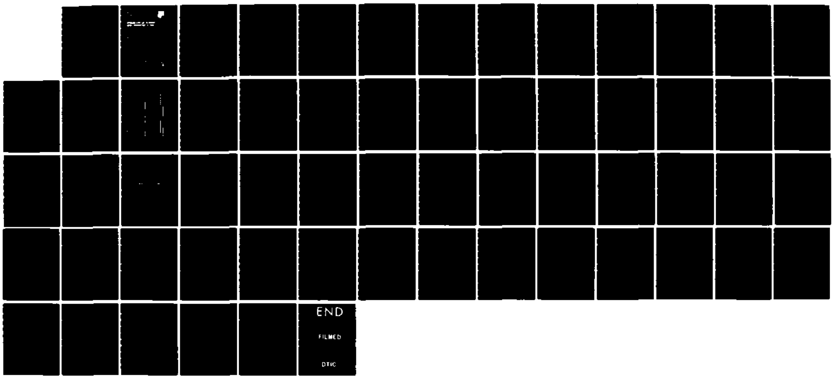
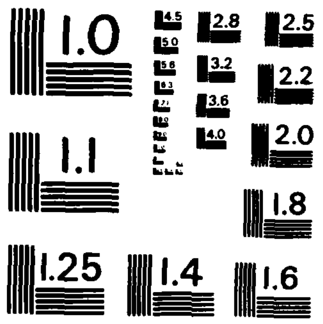

END
FILMED
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

12

IMPLEMENTATION AND REFINEMENT OF DESIGN SPECIFICATIONS IN PEGASYS

SRI International

Sponsored by
Defense Advanced Research Projects Agency (DOD)
ARPA Order No. 4130

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

RADC-TR-84-128 has been reviewed and is approved for publication.

APPROVED: [illegible]
DOUGLAS A. WHITE
Project Engineer

APPROVED: [illegible]
RAYMOND P. URTZ, JR.
Acting Technical Director
Command and Control Division

FOR THE COMMANDER: [illegible]
DONALD A. BRANTINGHAM
Plans Office

REPRESENTATION AND REFINEMENT OF VISUAL SPECIFICATIONS IN PEGASYS

Mark S. Moriconi

Contractor: SRI International
Contract Number: F30602-81-K-0176
Effective Date of Contract: 1 May 1984
Contract Expiration Date: 29 February 1984
Short Title of Work: Graphics Oriented Environment
Program Code Number 3D20
Period of Work Covered: May 81 - Feb 84

Principal Investigator: Dr. Mark S. Moriconi
Phone: (415) 859-5364

Project Engineer: Douglas A. White
Phone: (315) 330-2748

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Douglas A. White (COES), Griffiss AFB NY 13441 under Contract F30602-81-K-0176.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A148733

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S): RADC-TR-84-128
6a. NAME OF PERFORMING ORGANIZATION: SRI International Computer Science Laboratory; 6b. OFFICE SYMBOL (If applicable):	7a. NAME OF MONITORING ORGANIZATION: Rome Air Development Center (COES)
6c. ADDRESS (City, State and ZIP Code): 333 Ravenswood Ave Menlo Park CA 94025-3493	7b. ADDRESS (City, State and ZIP Code): Griffiss AFB NY 13441
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Defense Advanced Research Projects Agency; 8b. OFFICE SYMBOL (If applicable): IPTO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F30602-81-K-0176
8c. ADDRESS (City, State and ZIP Code): 1400 Wilson Blvd Arlington VA 22209	10. SOURCE OF FUNDING NOS.: PROGRAM ELEMENT NO. 61101E; PROJECT NO. D139; TASK NO. 19; WORK UNIT NO.

11. TITLE (Include Security Classification): REPRESENTATION AND REFINEMENT OF VISUAL SPECIFICATIONS IN PEGASYS

12. PERSONAL AUTHOR(S): Mark S. Moriconi

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM May 81 TO Feb 84	June 1984	

16. SUPPLEMENTARY NOTATION: N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Specification, computer programming, logical representation, interactive computer graphics, program design methodology, design refinement
09	02		

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

This report describes techniques for the representation and refinement of visual specifications in the context of PegaSys (Programming Environment for the Graphical Analysis of SYStems), a system that supports a visual paradigm for the development and explanation of large software designs. Visual specifications are pictorial, mostly non-textual, descriptions of interactions among conceptual entities in a system design. Pictures have a computational meaning that is represented in a formal language, called the *form calculus*. The form calculus is extensible in that it contains a core set of primitives which can be used to build a variety of abstract design models. Complexity is managed by means of picture hierarchies, whose construction is guided by a precise refinement methodology.

The representation and refinement techniques presented ~~here~~ have been implemented and all reasoning is fully automatic and efficient. Determining the validity of a picture refinement, for example, involves either the application of a single graph algorithm or the

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED [✓] SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Douglas A. White	22b. TELEPHONE NUMBER (Include Area Code): (315) 330-2748; 22c. OFFICE SYMBOL: RADC (COES)

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

cont

proof of a formula whose predicates range over small, finite sets. Excerpts from a sample session with PegaSys are used to illustrate a hierarchy of visual specifications.

Originator-supplied Keywords [illegible] — (see [illegible])

A-1

CONTENTS

Acknowledgments

The form calculus was developed in collaboration with Amy Lansky and the version of PegaSys described here was implemented by Dwight Hare. The work has also benefitted from the comments, suggestions, and efforts of Ed Ashcroft, Alain Conchon, David Plaisted, Leslie Lamport, Stan Rosenschein, and Fritz Vogt.

The aspects of this research dealing with hierarchies was funded in part by the Office of Naval Research under Contract N00014-83-C-0300.

1. Introduction

Much of the high cost of software can be attributed directly to the inadequacy of system documentation and the tools for generating and manipulating it. This inadequacy especially impacts software maintenance, which, according to many studies, accounts for most of the life-cycle cost of a system. Regardless of the documentation language used, formal system documentation has tended to be difficult to understand. One reason for this is the use of unfamiliar specification constructs. Another is the absence of explicit information about interactions between different parts of the documentation or between different parts of the actual system code. Unfortunately, it is impractical, if not impossible, to generate a comprehensible description of system interactions from final documentation or code. Interactions should be described in terms of the abstractions used in their conceptualization; most often, neither documentation nor code directly mirrors (or should directly mirror) these abstractions.

What is needed, then, is a formal language for explicitly describing system interactions that is easy to understand and use, yet is rich enough to express important interactions at multiple levels of abstraction. Ideally, this language would be supported by a system capable of recording the hierarchy of refinements which led to the interactions in the final design and, most importantly, of ensuring that each refinement step is methodologically sound.

Our approach to these problems involves the use of visual (graphical) specifications of the "form" of a system — that is, the important conceptual entities of a design and how they interact. Because of their intuitive appeal, pictures have been used extensively by computer scientists in textbooks, professional publications, and on blackboards to explain the form of a system. In the past, however, such uses have tended to be quite imprecise as a means of documentation, resulting in pictures that are confusing and easily misinterpreted. For example, the same graphic symbol is often used to represent a process, a subprogram, and a data structure, all in the same picture. Similarly, the same arrow might represent the flow of data to a process, the flow of control between subprograms, or the writing of data into a data structure, all quite distinct concepts.

This paper describes a technical basis for the use of pictures as formal, machine-processable documentation. In particular, we discuss:

- **Picture Representation**. Pictures must be treated as structures in a language for describing properties of computer programs, not as bitmaps or graphical structures devoid of computational meaning. To this end, we have developed a language to encode the meaning of pictures, called the *form calculus*, which contains a small set of primitive predicates for describing entities and interactions useful in conceptualizing a design. Sentences in the form calculus are called *forms*. The form calculus is extensible in that this core of primitive predicates can be used to build more abstract notions about the form of a design. It is possible to define, for example, both a dataflow model of form (with asynchronous processes communicating by value passing) and a von Neumann model (which uses stored, possibly shared, variables).
- **Picture Refinement**. We have also developed a picture refinement methodology and precise rules for determining whether it has been applied properly. The methodology only allows refinements that preserve certain important properties dealing with, among other things, transmission of data, logical consistency of interactions, and the use of names and values. It does this by means of machine-enforceable rules which take into account both the syntax and semantics of pictures. The methodology is extensible in the sense that it is possible to introduce specialized restrictions on how newly defined concepts can be refined.

PegaSys supports the structured composition and refinement of pictures in the form calculus. It automatically checks the syntactic and type consistency of entities and relationships within a picture, as well as the adherence of each picture refinement to the requirements of our picture refinement methodology. Needed proofs are done quickly and without human intervention because all formulas to be proved involve predicates whose variables range over small, finite domains.

Eventually, we expect PegaSys to provide two important capabilities not discussed in this paper. The first concerns the connection of a hierarchy of visual specifications to system code. Except in the simplest cases, there appears to have been few attempts to verify the consistency between a specification of the form of a system and its actual form. The richness of the form calculus makes the requisite analysis considerably more difficult than, say, a control flow analysis. For example, one may define an information flow relation that takes into account indirect flows, or the possibility of aliased names.

The second concerns the use of animation to provide the user with an intuitive explanation of system behavior. In the past, the most common approach has been what might be called "language-based" animation, the use of *predefined* displays of structures (usually of data) appearing in the actual program text as the basis for what the user sees. The drawback of the language-based approach is the near impossibility of predefining good design abstractions for pictures. In contrast, our approach avoids the animation of the complete and intricate behavior of a system, and instead, presents only that information dictated by a user-defined picture. We call this a "content-based" approach, since the meaning of a visual specification is the basis for what is seen.

This paper is organized as follows. After reviewing related work, we present an overview of the current PegaSys system and an example of its use. Following that, we describe the initial solutions we have found for picture representation and refinement. This discussion includes two examples of *derived* models of computation (von Neumann and dataflow) which are constructed from the core primitives of the form calculus. The final section concludes with a discussion of how this work contributes to the area of software development and outlines some future and ongoing work.

2. Related Work

There has been a number of attempts to capture the form of a system and to explain its behavior in graphical terms.

- Previous attempts to develop formal visual specification languages have met with limited success, primarily because the languages were not expressive enough and the abstraction techniques were inadequate. Examples of formal visual specifications include flowcharts, dataflow diagrams (such as in [2], [3], and [6]), and structure charts [6]. A richer visual formalism is the plan calculus [5], which has been used primarily to represent standard programming knowledge, not the form of a system design. Our formalism differs from previous ones in that it supports at least two models of computation, a greater, yet unified, array of concepts for von Neumann-style descriptions, and the introduction of user-defined concepts.

- Complexity is typically managed by means of visual hierarchies, as in the dataflow hierarchies in [4] and the plan hierarchies in [5]. Refinements of form must normally satisfy simple connectivity constraints. Our refinement methodology, on the other hand, provides for more powerful notions of logical and form refinement. Moreover, while it provides generic constraints on all refinements, it is possible to introduce specialized refinement constraints. Examples of this are given in Section 6.1.2.

3. Overall Design of PegaSys

This section presents an overview of the entire system. The reader should note that only the parts dealing with representation and refinement, as well as the user interface, are fully implemented.

PegaSys is being implemented in Interlisp-D and runs on a Xerox 1100 (aka Dolphin) personal computer. Figure 1 shows its main data structures (denoted by "blobs") and sequential subsystems (denoted by rectangles), as well as important "information-flow" relationships between them (denoted by arcs).

The primary inputs to PegaSys are pictures and Ada source code. The user interface, which mediates all user interaction with the system, includes separate structure-oriented editors for constructing pictures and programs. Pictures are represented internally as forms and Ada programs as abstract syntax trees. The hierarchy manager is responsible for ensuring that each level in a picture hierarchy is a valid refinement of the next higher level and for supporting the structured perusal of a hierarchy. A perusal may follow steps in the design refinement history and may also take advantage of design "views", which group smaller subsets

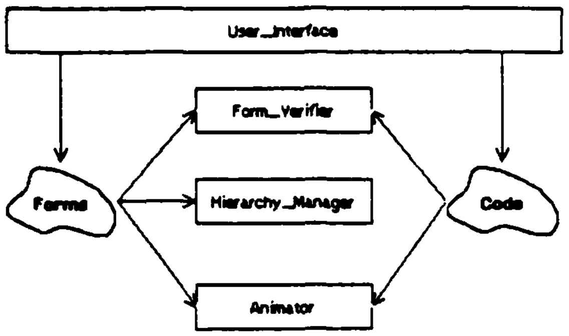


Figure 1: Architecture of PegaSys

of logically related graphic symbols. The form verifier will ensure that the picture hierarchy is logically consistent with the Ada code that it is intended to describe. The animator will explain the dynamic execution of an Ada program in terms of forms.

There are at least three important characteristics of the overall design. The first is that pictures always are treated as computationally meaningful objects. They are never considered simply as bitmaps or as graphic structures devoid of computational meaning. This property manifests itself in the design of every system component. For example, PegaSys' picture editor enforces constraints on picture construction which correspond to the syntactic and type constraints of the underlying form calculus. If graphic symbols are arranged in such a way as to denote a property that is not computationally meaningful, an error message is given.

The second important characteristic concerns the user interface. Interaction with PegaSys takes place in terms of pictures, not the internal logic of the form calculus. This means, for example, that the process of visual specification has been designed to allow all reasoning about pictures to occur automatically and efficiently. The technical implication of this, as explained in the next section, is that specifications state *potential*, instead of actual, relationships.

The last key characteristic of PegaSys is that its internal representation and manipulation of the meaning of pictures (in the form calculus) is independent of specific graphic conventions or textual languages. If graphical conventions are changed, only the picture editor need be modified. If a specification or programming language other than Ada were to be described by pictures, only those aspects of the system that deal with the semantics of the language would have to be recoded.

4. An Example Scenario

Figure 2 illustrates a use of the PegaSys refinement methodology. Starting with the window in the upper left-hand corner and moving clockwise, we depict the construction and refinement of the form of a distributed communication protocol intended to achieve reliable message transfer over an unreliable transmission line. We refer to this example, and explain it in full detail, in subsequent sections.

Figure 2a depicts the protocol as a high-level network service. A source and destination process send messages to and receive messages from a network communication layer. In order to refine the network layer, the user positioned the cursor within the ellipse labeled *Network_Layer* and pressed a button on the mouse. This selects the associated predicate in the underlying form. The user then constructed a picture and told PegaSys that it was intended to be a refinement of the selection. The result is Figure 2b, in which the network layer has been refined into a data link service. (The "sockets" with numbers specify the correspondence between pictures.) Messages from the source are sent to a sender process which communicates directly with the data link layer. Similarly, messages received from the data link layer are handled by a receiver process before being passed on to their destination.

Note that PegaSys found Figure 2b to be a valid refinement of the network layer. This analysis, in general, is based on logical, as well as methodological, considerations.

In Figure 2c, the window in the lower right-hand corner, the data

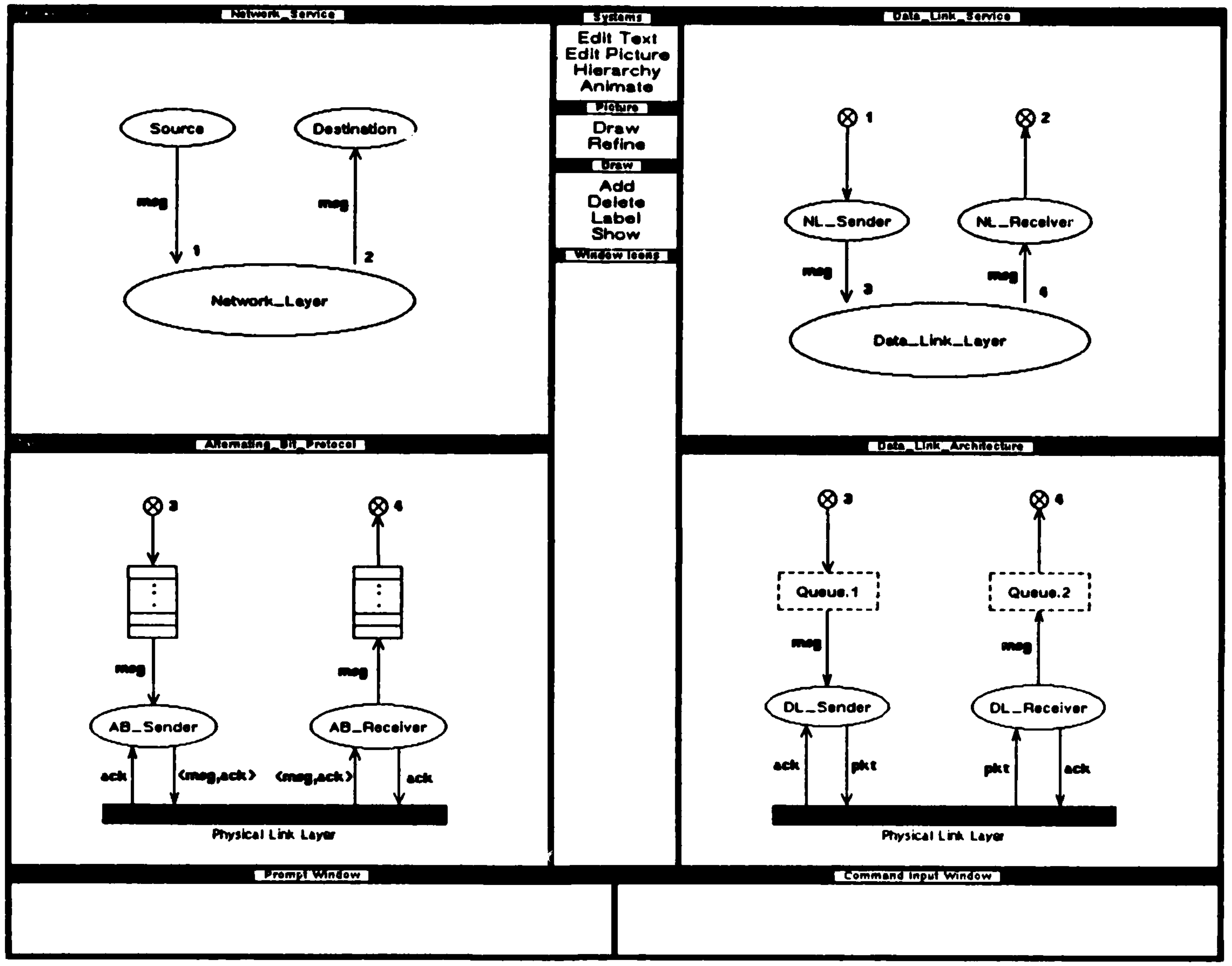


Figure 2: An example visual specification hierarchy, beginning in the upper-left "window" (a) and progressing clockwise (b)-(d).

link layer has been refined into a picture that includes the actual physical link. Messages from the network-layer sender are buffered by a queue. A data-link sender takes messages from the queue and interacts with the physical link layer via packets and acknowledgments. Similarly, a data-link receiver process communicates directly with the physical link layer. Once messages have been received, they are buffered before being transmitted to the network-layer receiver.

Finally, in Figure 2d, the queues have been refined into icons representing data abstractions. (See Figure 6 and Section 5.3 for an explanation of how the queues were refined.) *DL_Sender* and *DL_Receiver* have been renamed to be *AB_Sender* and *AB_Receiver* (to suggest that the alternating bit protocol is used to transmit messages over the unreliable line). Packets have been further defined as sequences consisting of two elements, a message and an acknowledgment.

It is understood that these pictures, as well as any other pictures representable in the form calculus, specify *potential* relationships. For example, an informal interpretation of Figure 2a is that messages flow from process *Source* to process *Network_Layer*. "Uncertainty" in the interpretation of relations stems from the mathematical undecidability of the primitive relations in the form calculus when interpreted with respect to actual executions of source code. In other words, we can only state that messages *might* flow, but not that the *will* flow. In fact, any reasonable set of relations for specifying the form of a system would have the same characteristic. Although this notion of potentiality should always be kept in mind, we henceforth describe specifications as though they express "certainties".

5. Representing the Meaning of Pictures

Our approach to the use of pictures separates the computational meaning of a picture from how it is expressed graphically on a display. The computationally important aspects of a picture are represented internally by a *form* — a sentence in a simple logic. The entities and predicates in a form may be *primitive* or *derived*.

The primitive predicates used in forms were chosen to be suitable for describing a low-level von Neumann model of computation. We attempted to identify concepts that:

- Are precise enough to avoid multiple or unintended interpretations of a picture.
- Easily compose to describe useful, higher-level concepts about the form of a system.
- Do not bias the way in which a specification is realized by an implementation.

A small set of primitive concepts that satisfy these goals were chosen (seven entities and seven interactions). These concepts appear to be sufficient for describing a wide range of useful models of form.

The form calculus is extensible in the sense that new notions (derived predicates) can be defined in terms of existing ones. It is possible to define not only more elaborate von Neumann models, but also conceptually different models, such as dataflow. The ability to represent the form of many models of computation is crucial to a flexible design system. For example, it is often convenient to conceptualize a system design in terms of dataflow initially, and then refine that conception into a von Neumann-style description of an imperative program.

The cosmetic (computationally unimportant) aspects of a picture are represented by a graphic structure consisting of graphic symbols and their characteristics, such as size and location. The graphical and logical representations of a picture are connected so that manipulations of the graphic structure can be related to the associated form, and vice versa.

The representation of a picture as two separate, but connected, structures has three major benefits. The first is that the underlying form calculus can be used to guide the construction of pictures in much the same way that structured editors guide the construction of programs. Secondly, cosmetic changes to a picture do not require internal update of the associated form and, therefore, no reasoning need be done to determine whether the change was logically correct. An example of a cosmetic change is an adjustment to the size or location of a graphic symbol. Lastly, changes in display conventions do not require any recoding of the logical machinery for representing and reasoning about the computational meaning of pictures.

We now describe the basic lexical structure of forms. We then present the core von Neumann model and illustrate how it can be used to define a derived von Neumann model and a derived dataflow model.

5.1. Lexical Structure of Forms

A form is a finite conjunction of predicates on the elements of a finite set of symbols. Unary relations denote the types of conceptual entities in a design, symbols (constants) denote particular instances of these entities, and non-unary relations denote relationships among instances. Different instances must be denoted by distinct constants.

A simple example of a form, corresponding to the picture in Figure 2a, is the following:

$$\begin{aligned}&process(Source) \;\wedge\; process(Destination) \;\wedge\\ &process(Network\_Layer) \;\wedge\; type(msg) \;\wedge\\ &DataFlow(Source, Network\_Layer, msg) \;\wedge\\ &DataFlow(Network\_Layer, Destination, msg)\end{aligned}$$

This form represents three different "process" entities, one "type" entity (representing a set of possible values), and two "dataflow" relations between entities. The type *msg* is used as an argument of the *DataFlow* predicate to indicate that data of type *msg* is transmitted between processes.

Constraints on the set of relations allowed in a form restrict how entities may fit together. Associated with every non-unary relation R is an ***acceptability constraint***, a first-order formula, that must be satisfied before R can be added to a form. Intuitively, an acceptability constraint provides strong typing constraints on the entities related by R. For example, suppose that we want to restrict the use of the relation $DataFlow(x, y, d)$ so that it can only be applied to processes. This is expressed by the acceptability constraint $process(x) \wedge process(y)$.

An acceptability constraint is checked by means of a logical proof. A form $\mathcal{F}$ is a ***legal form*** if and only if, for every relation R in $\mathcal{F}$, the formula $\mathcal{F} \supset R_A$ is true, where R_A is the acceptability constraint for R. Henceforth, when we use the term "form", we mean "legal form" unless stated otherwise.

Let $\mathcal{F}$ denote the form for Figure 2a. In order to check the type constraints for *DataFlow(Source,Network_Layer,msg)*, PegaSys would prove

$$\mathcal{F} \quad \supset \quad process(Source) \land process(Network\_Layer)$$

If the proof fails, either *Source* or *Network_Layer*, or both, is of the wrong type.

The truth (or falsehood) of such formulas is easily determined. Most often, each predicate of an acceptability constraint is an explicit premise belonging to $\mathcal{F}$. In other cases, the proof of acceptability may involve quantification over entities of $\mathcal{F}$. But, since the number of entities is always finite and relatively small, all possibilities can be enumerated very quickly.

Notice that there is a direct mapping between the pictures in our scenario and their forms. Intuitively, a form describes a finite, directed graph, whose nodes and edges have "kind" and "label" properties. Each unary relation is represented by a node whose label property is a symbol and whose kind property is the relation; a non-unary relation is represented by an edge, whose label property is a symbol denoting transmitted data and whose kind property is the relation. For example, the relation *process*(*Source*) is depicted by a node with label property *Source* and kind property *process*. *DataFlow(Source,Network_Layer,msg)* is depicted by an edge from *Source* to *Network_Layer* with label *msg* and kind *DataFlow*. In the figures, different node shapes (such as an ellipse or a rectangle) denote different kinds of nodes. Edge annotations are used to denote the kind property of edges. For example, an edge annotation *d* may be used as an abbreviation for relation symbol *DataFlow*. Although these annotations were suppressed in our scenario, they can be made visible by pressing a button on the mouse.

It should be pointed out that it is possible, and sometimes useful, to define derived concepts that suggest a visual presentation other than graphs. For instance, a relation among three entities cannot be represented, at least directly, by the graph model just described. In such situations, the present implementation of PegaSys displays the relations as text.[1]

[1]In fact, primitive relations *declare* and *aliased* of the core von Neumann model are displayed as text.

dataAbs:	Denotes an instance of a data abstraction.
type:	Denotes a set of possible values.
name:	Denotes the name of a data object which may contain a value of a given type.
value:	Denotes an element of some domain.
tuple:	Denotes a sequence of data objects.
process:	Denotes an entity whose execution may proceed in parallel with other processes.
subprogram:	Denotes the set of sequentially executed actions within a procedure or function.

Figure 3: Primitive unary relations for conceptual entities.

5.2. Core von Neumann Model

The von Neumann model of computation has two intrinsic characteristics, both of which are reflected in imperative programming languages. First, it has an updatable memory which is manifested in programs by the use of stored variables. Secondly, it has an instruction counter, which is manifested in programs by a rigid notion of transfer of control. The following describes how the primitives of the form calculus account for these two concepts. The notions of "control" and "data" have been formulated generally enough to allow derived models to be formed which do not utilize stored variables or a von Neumann notion of control. An example of such a model is the dataflow model.

In describing the core model, we will find it useful to distinguish between two kinds of entities. *Active entities* are entities which may access or modify a data object; *passive entities* are transmittable entities that describe properties of an unprotected data object. We begin by describing data objects (both active and passive) and their role in specifications. All of the core concepts described below are summarized in Figures 3 and 4.

P1. declare (x, y) : name $(x) \wedge$ type (y)

P2. signal (x, y) : operation $(x) \wedge$ process (y)

P3. control (x, y) : operation $(x) \wedge$ subprogram (y)

P4. returnOfControl (x, y) : subprogram $(x) \wedge$ operation (y)

P5. modDataOf (x, y, n) : operation (x) $\wedge$
(operation $(y) \vee$ dataAbs (y)) $\wedge$ name (n)

P6. aliased (x, y) : name $(x) \wedge$ name $(y) \wedge x \neq \epsilon \wedge y \neq \epsilon$

P7. accessDataOf (x, y, v) : operation (x) $\wedge$
(operation $(y) \vee$ dataAbs (y)) $\wedge$ value (v)

Figure 4: Primitive non-unary relations.

5.2.1. Data Objects

An instance of a data abstraction is denoted by the unary relation *dataAbs* and represents an "encapsulated" data object. Examples of its realization in a programming language are the "class" in Simula and the "package" in Ada. Encapsulation implies an explicit separation of the concrete realization (implementation) of a data object from its use in a program. The data objects within a data abstraction may only be accessed through a set of specified operations. Thus, each data abstraction instance functions as an *active entity* with a controlled interface between itself and the rest of a program. The queue in Figure 2c is an example of a data abstraction. To reflect the fact that it is an active entity, it is displayed as a separate node in the graph, as opposed to a label on an arc.

In contrast, the properties of passive data objects (variables) may be *directly* accessed, modified, and transmitted. A passive data object is characterized by three properties:

- A *type* denotes a set of values and a set of associated operations. If t is a type, $type(t)$ is true.
- A *name* is used to refer to the data object. The predicate $name(n)$ is true if n is a name. Names are needed in order to control access to data objects.
- A *value* is an element of some domain. If v is a value, $value(v)$ is true. If a data object has type t, the value of the object must be an element of the domain denoted by t. We henceforth use the notation "$n.val$" to denote a value of a data object with name n. Specific values (e.g., "0", "abc", or "true") are not used in forms. Such values would be needed to specify what a system is intended to do, i.e., its behavior; in describing form, we need only "generic values" such as $n.val$.

A special relation is employed to explicitly state that a particular name and type is associated with the same data object. The binary relation *declare(n,t)* specifies that the object with name n is "bound to" type t. It implies that $n.val$ has type t.

The entities denoted by these unary predicates are called passive entities because they characterize properties of unprotected data. Passive entities are used by specifications to describe and transmit information about a data object. Sometimes, we shall want to indicate the tranmission of a passive entity without identifying a *particular* one. In such cases, we use ϵ, the "empty" datum. An example of its use is $DataFlow(a, b, \epsilon)$. All three relations (*type*, *name*, and *value*) are satisfied by ϵ.

It is desirable, at times, to provide a more structured description of a data object. This is done by means of ordered tuples, each element of which represents a different data object. For example, the type *pkt* in Figure 2c is refined to a tuple of types $\langle msg, ack\rangle$ in Figure 2d. This refinement indicates that a packet consists of two components, a message and an acknowledgment.

5.2.2. Operations

The form calculus contains two types of primitive active entities that manipulate data objects — processes and subprograms. A process, denoted by the unary relation *process*, may be thought of as an entity operating concurrently with every other process entity. It consists of a series of sequentially executed actions, including those occurring as a result of subprogram invocation. (Note, however, that forms do not include information about the identity or order of the actions within a process; they only specify its relationships with other entities and the identity of the data objects it modifies and accesses.)

A subprogram entity is denoted by the unary relation *subprogram*, which can be thought of, in programming language terms, as a procedure or a function. Actions within a subprogram can result in communication with a process, another subprogram, or itself (in the case of recursive subprograms).

We will use the derived unary relation *operation* as shorthand for an entity which satisfies either the process or the subprogram relation.

5.2.3. Control

In general, the pure notion of "transfer of control" refers to communication between active entities in which there is no *explicit* transfer of data. Two examples of this in the von Neumann model are the "signaling" of a process or the transfer of control to a parameterless subprogram.

The form calculus primitives separate the notion of how control flows in a program from that of how data flows. As seen later, this makes it easy to define derived relations that mix the two in various ways.

There are three primitive relations in the form calculus for describing transfer of control. One describes communication between two threads of control; the other two describe control transfer to a called subprogram (within the same thread of control). The primitive relation for transferring control from an operation to a process is *signal*(x, y) (see P2 of Figure 4). It says that operation x attempts to communicate with process y; it does not indicate whether data is transmitted. Note that x may be a subprogram or a process.

Control is transferred to a subprogram by means of the *control* relation (P3). This type of control transfer may be initiated by a process or a subprogram; recursive subprograms pass control to themselves. The return of control from a subprogram to the point of transfer is denoted by the *returnOfControl* relation (P4). Using two relations to model transfer and return of control, rather than two instances of the same "control" relation, avoids possible misinterpretations. First, using *control* to describe both transfer and return of control would suggest that they are the same. Transfer of control to a subprogram always initiates execution at the beginning of the subprogram, while return of control from a subprogram resumes execution at the point of transfer. A second possible misinterpretation concerns the role of processes. A process can initiate this type of control transfer to a subprogram, but not vice versa. Using *control* to describe return of control would suggest that subprograms could initiate this type of control transfer to a process. This would be inconsistent with our intuitive notions about the role of processes and subprograms.

As seen later, derived "subprogram call" relations may be defined by combining *control* and *returnOfControl*. However, since we have separated the notions of control and return of control, it is possible to define specialized derived relations that involve only one of them.

5.2.4. Manipulation of Data Objects

There are two possible kinds of interaction with data — modification (writing) and access (reading). Data that is shared between processes and subprograms may be represented as (passive) unprotected variables or as (active) data abstractions.

The relation *modDataOf* (see P5) is used to specify *all* modifications to data. When the shared data is an unprotected variable, the relation *modDataOf(x,y,n)* says that operation x modifies the value of a passive data object with name n belonging to operation y. Note that, when we interpret this relation with respect to an actual programming language, n may be a formal parameter of x or a local variable of x (in which case $x = y$), or nonlocal to x (in which case $x \neq y$).

Derived relations can be used to define special kinds of interaction with data by placing restrictions on any combination of x, y, or n. An example is the notion of "side effect", which can be specified by the restriction $x \neq y$. In practice, a side effect can occur as a result of a modification to a data object transmitted by reference (i.e., a name was passed) or a modification to a nonlocal variable.

For a data abstraction, $modDataOf(x, y, n)$ specifies that an operation x modifies some variable n associated with data abstraction y. As seen later in Section 6.1.2, x must be an operation explicitly associated with protected data object n in abstraction y. In Ada, for example, x would be an operation in package y. Special refinement constraints ensure that other operations do not directly access data within y.

The *modDataOf* relation does not account for the fact that modification of data can have *indirect* effects due to aliasing of names. Aliasing occurs when two different names refer to the same data object. The *aliased* relation (P6) is a symmetric relation between names. We show later how the aliased relation may be used in defining a derived predicate expressing the notion of modification via aliasing. That is, if an object with name n is modified, and $aliased(m, n)$, then an object with name m may have been modified as well.

Simple access to data is specified with the relation $accessDataOf(x, y, v)$ (see P7) which says that x accesses a data value v belonging to y. Just as with *modDataOf*, we consider local and nonlocal access to data, as well as shared variables and data abstractions. For unprotected shared variables, $accessDataOf(x. y. v)$ says that operation x accesses the values v belonging to operation y. If y is a data abstraction, x accesses a value v belonging to abstraction y.

5.2.5. Naming and Scope

The linguistic details of naming and scope are handled in a straightforward fashion. First of all, we avoid the problem of handling duplicate names (symbols) within different scopes by requiring that all unique entities have unique names. This does not preclude a more elaborate naming structure in the actual implementation, since different names in forms need not be associated with different names in programs.

As an aid to the user, unique names can constructed automatically by PegaSys in certain situations. For the purposes of this paper, assume that this is done in only two situations. Local variables are qualified by the name of their "owner". For example, $x.n$ denotes the unique name of data object n belonging to entity x. PegaSys may also generate unique names for instances of data abstractions, such as $Queue.1$ and $Queue.2$.

5.3. A Derived von Neumann Model

Derived relations are defined using first-order logic with equality. Variables must range over finite domains, in particular, the entities and relations in a form. Every derived relation has an *acceptability constraint*, as defined earlier, and a *definition* of the form $R \equiv P$, where R is a new relation and P is a formula containing only existing relations. As explained later, definitions have several uses. For example, they are used in determining how a derived relation can be refined into more primitive relations. For example, a use of $SimpleCall(a,b)$ could be refined into the relations $control(a,b)$ and $returnOfControl(b,a)$ if $SimpleCall(x,y)$ is defined to be $(control(x,y) \land returnOfControl(y,x))$ (see B1, Appendix B).

The derived relations employed in the scenario are contained in Figure 5 and explained below; examples of other useful derived relations can be found in Appendix B. We have already seen a derived unary relation, namely, *operation*.

The relation in our derived von Neumann model for expressing unidirectional communication with a process is $vDataFlow(x,y,d)$ (see D1 in Figure 5). Its acceptability constraint allows both subprograms and processes to communicate with a process. Communication may involve the transfer of values or names of shared data. Its definition (the sec-

D1. vDataFlow (x, y, d) :

- operation $(x) \land$ process $(y) \land$ (value $(d) \lor$ name (d))
- $[\, d = \epsilon \supset$ signal $(x, y)\,] \land$
  $[\, d \neq \epsilon \supset$ signal $(x, y) \land$ (accessDataOf $(y, x, d) \lor$ modDataOf (y, x, d)) $]$

D2. Read (x, y, v) :

- operation $(x) \land$ value $(v) \land$
  $[$ (dataAbs $(y) \land (\nexists z)(\nexists d)$ [modDataOf $(z, y, d) \lor$ accessDataOf (z, y, d)]) $\lor$
  (operation $(y) \land (\exists dt)$ ReadChain (y, dt)) $]$
- accessDataOf (x, y, v)

D3. Write (x, y, n) :

- operation $(x) \land$ name $(n) \land$
  $[$ (dataAbs $(y) \land (\nexists z)(\nexists d)$ [modDataOf $(z, y, d) \lor$ accessDataOf (z, y, d)]) $\lor$
  (operation $(y) \land (\exists dt)$ WriteChain (y, dt)) $]$
- modDataOf (x, y, n)

D4. DataFlow (x, y, t) :

- process $(x) \land$ process $(y) \land$ type (t)
- signal $(x, y) \land$ accessDataOf (y, x, t)

Figure 5: Derived relations employed in scenario.

ond formula at D1) says that communication takes the form of a signal, possibly coupled with data transmission.

Next, we consider two derived relations describing interactions with data abstractions. The pictures in Figures 2c and 2d illustrate the use of these relations. There are at least two ways of using data abstractions. At a very abstract level, data types may simply be "read" or "written". At a lower level, we explicitly identify the operations that have sole direct access to the protected data. Once these operations have been explicated, direct "reading" or "writing" of protected data may not occur.

For example, the picture in Figure 2c says that sender process *NL_Sender* "writes" into a queue. In Figure 6, the queue has been refined into a set of operations having exclusive access to an data abstraction. At this level of abstraction, the *Enq* and *Deq* operations are seen as operations which manipulate an asynchronous, first-in-first-out queue. The entire system refinement is depicted in Figure 2d, where the user chose to display the queue icon, rather than the queue replacement form of Figure 6.[2]

D2 and D3 define the read and write relations. Notice that the acceptability constraints never allow users of a data abstraction to bypass the operations associated with it. For example, if we have $Write(x, y, v)$, y can be a data abstraction only if no operations have been specified that directly access or modify y. In fact, we can have $Write(x, y, v)$ only if there is some chain of writes between y and a data abstraction, terminating in a write or a direct modification of the data abstraction. This is captured by predicate $WriteChain(y, dt)$, which is defined as follows:

$$(\exists\, z_1, \ldots, z_n, d_0, \ldots, d_n)\, [\, \text{Write}\,(y, z_1, d_0) \wedge \text{Write}\,(z_1, z_2, d_1) \wedge \\ \text{Write}\,(z_2, z_3, d_2) \wedge \ldots \wedge (\text{Write}\,(z_n, dt, d_n) \vee \text{modDataOf}\,(z_n, dt, d_n))]$$

As explained in Section 6.1.2, *WriteChain* allows us to define hierarchies of data abstractions. The analagous restrictions must hold for *Read*.

The definitions of *Read* and *Write* state that they are equivalent to

[2]The queue icon was created by the user as a bitmap. We are in the process of building a library of standard data abstractions with associated icons. These may be connected to actual instances of a data abstraction for animation purposes. However, we have not yet extended the form calculus to allow such icons to be treated as formal objects.

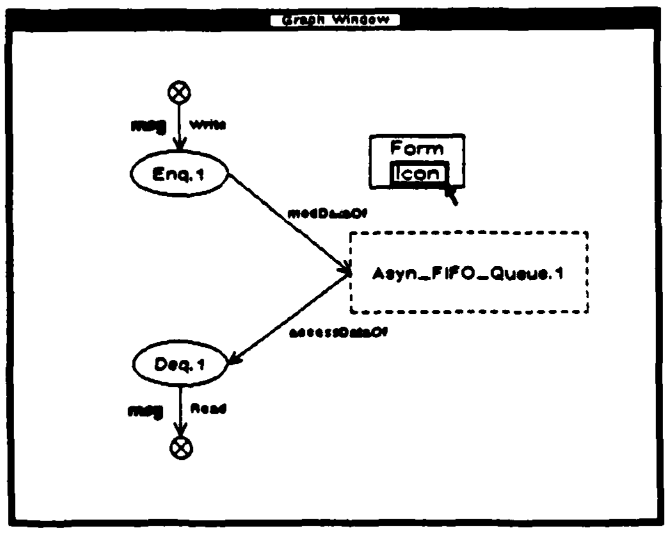


Figure 6: Replacement form for *Queue*.1.

accessDataOf and *modDataOf*, respectively.

5.4. A Derived Dataflow Model

The dataflow model encourages one to think about a problem in terms of data flowing from one functional entity to another. Each of these entities may be viewed as operating concurrently with every other entity, and can be understood independently of other entities as well. Enabled entities consume input values, execute, and produce a set of output values for use by other entities. In line with standard dataflow philosophy, a functional entity cannot have side effects. A good description of dataflow models can be found in [1].

A dataflow program can be thought of as a graph, where functional entities are denoted by nodes and data is viewed as flowing on arcs from one node to another. This is represented by the relation *DataFlow*(x, y, t) (see D4 of Figure 5) which says that process x communicates with process y by transmitting values of type t. The *DataFlow* relation does not bias the choice of communication mechanism (synchronous or asynchronous) used in an underlying system implementation.

Transmitted data is specified as a type. For example, *DataFlow(Source,Network_Layer,msg)* says that values of type *msg* (messages) flow from *Source* to *Network_Layer*. Unlike the von Neumann model, there is no concept of names (since there is no updatable store). Notice that *DataFlow* is a special case of von Neumann data flow in which x will always be a process and the *modDataOf* relation will never be satisfied.

Finally, we point out that this derived dataflow model may provide a useful conceptual tool for high-level design that is quite distinct from our von Neumann models. For example, dataflow specifications omit details of data storage and access. The refinement techniques described in this paper only partially accomodate the transition from a dataflow specification to a von Neumann-style specification. Ongoing research in passive entity refinement techniques should resolve the remaining problems. Note, however, that our scenario does illustrate a particular transition between the two models (see Appendix A).

6. Picture Refinement Methodology

A design consists of a hierarchy of levels, where each level is a *complete* description of the form of a system at a particular level of detail. A level is formed by a sequence of refinements to the immediately preceding level in the hierarchy. Hence, a design can be described as a sequence

$$l_1 \quad r_1 r_2 \ldots r_m \quad l_2 \quad \ldots \quad l_n$$

where each l_i is a level and each r_i is the result of a refinement. Each l_i and r_i must be a legal form. A *legal refinement* must start with a legal form and result in a legal form. However, intermediate steps in a refinement may manipulate forms that are not legal.[3]

The methodology for constructing this hierarchy was designed to support the refinement of entities and interactions from the highest-level form to a form describing the actual implementation of a system. It has been carefully specified so that inappropriate refinements can be detected (automatically by the computer) by referring to refinement rules. Two kinds of form refinements have been particularly useful:

- **Active Entity Refinement.** An active entity may be replaced by a form, provided the replacement is done in such a way that preserves interactions involving the replaced entity.
- **Interaction Refinement.** An interaction may be replaced by more detailed interactions, provided that the interaction is a logical consequence of its replacement. This means that the interactions at different levels of a hierarchy must be logically consistent.

Note that, in our refinement model, both kinds of refinements *replace* something with something else.[4]

In PegaSys, a new level of a hierarchy is formed by first making a copy of the form at the previous level. Then, a series of replacements are made, the last of which completes the specification of the new level.

[3]Allowing only legal forms at all times would require that certain desirable refinements would be impossible or would have to occur in a particular order.

[4]This paper does not discuss passive entity refinement. Although we utilized an instance of passive entity refinement in the scenario (the replacement of *pkt* by the tuple $\langle msg, ack \rangle$), a more general methodology is presently under development.

The complexity of depicting a particular level can be managed by interactively constructed *views*, which are portions of a form. The scenario in Figure 2 contained views of four levels in the protocol design. The complete hierarchy and the refinements between levels are recorded by PegaSys, as seen in Appendix A.

6.1. Active Entity Refinement

Any refinement of an active entity must obey certain constraints. We begin by defining constraints that apply to *all* active entity refinements. Then, we explain how it is possible to introduce additional constraints for the purpose of enforcing a specialized refinement methodology.

6.1.1. General Procedure

Given a legal form $\mathcal{F}_1$, an active entity e in $\mathcal{F}_1$ may be replaced by a legal form $\mathcal{G}$ provided:

- The resultant form $\mathcal{F}_2$ is legal.
- Active entity e does not appear in $\mathcal{G}$.
- The replacement form "hooks up" with the original form in the same way that e did. That is, the resultant form was obtained by substituting an active entity of $\mathcal{G}$ for *each occurrence* of e in $\mathcal{F}_1$. Note that different occurrences of e may be replaced by different entities of $\mathcal{G}$.

Active entity refinement can best be illustrated by returning to our example. If we think of a form as a graph, the notion of preserving interactions reduces to that of preserving the connectivity of the graph. An example of this can be found at the beginning of the scenario, where *process(Network_Layer)* of Figure 2a was replaced in Figure 2b by the form

$$\begin{aligned}&\text{process}\,(NL\_Sender) \;\wedge\; \text{process}\,(Data\_Link\_Layer) \;\wedge\\ &\text{process}\,(NL\_Receiver) \;\wedge\\ &\text{DataFlow}\,(NL\_Sender, Data\_Link\_Layer, msg) \;\wedge\\ &\text{DataFlow}\,(Data\_Link\_Layer, NL\_Receiver, msg)\end{aligned}$$

and then connected to *Source* and *Destination* by

$$\text{DataFlow}\,(Source, NL\_Sender, msg) \;\wedge$$

DataFlow ($NL\_Receiver, Destination, msg$)

Observe that this is a legal form; *process(Network_Layer)* has been replaced, and the *DataFlow* relations preserve the connectivity of the graph in Figure 2a.

6.1.2. Additional Constraints

It is possible to further restrict the refinement of an active entity by means of an ***active entity refinement constraint.*** As a simple example, suppose that we require that an operation can only be refined into a process or a subprogram. This can be done by imposing the following constraint on the refinement of an entity e when *operation*(e) is true:

$$\begin{aligned}&\text{size}\,(\mathcal{G}) = 1 \quad \wedge \\ &(\exists\, x)\,[\;(\,\text{inForm}\,(\text{subprogram}\,(x), \mathcal{G}) \vee \text{inForm}\,(\text{process}\,(x), \mathcal{G})\,) \quad \wedge \\ &(\forall R)\,[\,\text{inForm}\,(R, \mathcal{F}_1) \supset \text{inForm}\,(R\,|^{e}_{x}, \mathcal{F}_2)\,]\,]\end{aligned}$$

where $inForm(R, \mathcal{F})$ means that relation R is in form $\mathcal{F}$ and $R\,|^{y}_{x}$ denotes a relation where every occurrence of y in R is replaced by x. $size(\mathcal{F})$ denotes the number of relations (conjuncts) in form $\mathcal{F}$.

The refinement of data abstraction entities must also follow certain specialized rules. In particular, refinement must preserve the integrity of encapsulated data by guaranteeing that only explicitly designated operations have access to it. In addition, conventions used by the unique name generator of PegaSys guarantee that each instance of a data abstraction has a unique name, and is associated with operations with related unique names. For example, in Figure 2d, the two queue abstractions are identical in form, except for naming conventions, to the abstraction shown in Figure 6. For instance, queues *Asyn_FIFI_Queue*.1 and *Asyn_FIFO_Queue*.2 are associated with operations *process*(*Enq*.1) and *process*(*Enq*.2), respectively.

These naming decisions and the derived relations in Figure 5 encourage a particular paradigm for data abstraction refinement. Figures 2c, 2d, and 6 provide an example of this refinement technique. We begin in Figure 2c with the relation

Read ($DL\_Sender, Queue.1, msg$)

Next, in Figure 6, *Queue.1* is replaced by

$$\begin{aligned}&\text{dataAbs}\,(\mathit{Asyn\_FIFO\_Queue.1}) \land \text{operation}\,(\mathit{Enq.1}) \land\\ &\text{operation}\,(\mathit{Deq.1})\land\\ &\quad\text{modDataOf}\,(\mathit{Enq.1}, \mathit{Asyn\_FIFI\_Queue.1}, \mathit{msg}) \;\; \land\\ &\quad\text{accessDataOf}\,(\mathit{Deq.1}, \mathit{Asyn\_FIFO\_Queue.1}, \mathit{msg})\end{aligned}$$

and then connected to our original form by

$$\text{Read}\,(\mathit{DL\_Sender}, \mathit{Enq.1}, \mathit{msg})$$

Note that the refinement rule for *Read* does not allow *DL_Sender* to read the queue directly now, because of the presence of *Enq* and *Deq* and their direct access to the queue.

This refinement illustrates the way in which the notion of "reading" an abstract data object can be refined into one of using a data object by means of its associated operations. This paradigm can be applied recursively. For example, if the asynchronous queue is to be implemented by a list abstraction, the queue would be replaced by a form containing a list data abstraction and some list operations. However, the original users of the queue would still regard the queue operations as the interface to the data object, even though it is now represented as a list. This indirect "chain" of reading or writing results in a legal form because of the predicate *WriteChain* (*ReadChain*) in the acceptability constraints for *Write* (*Read*).

These notions are captured by the following active entity refinement constraint. For a data abstraction entity e,

$$\begin{aligned}&(\forall dt)\,[\,\text{inForm}\,(\text{dataType}\,(dt), \mathcal{G}) \quad \land\\ &\qquad (\exists\, op, R)\,[\,\text{inForm}\,(\text{operation}\,(op), \mathcal{G}) \land \text{inForm}\,(R(\ldots op \ldots dt \ldots), \mathcal{G})\land\\ &\qquad\qquad (R(\ldots op \ldots dt \ldots) \supset\\ &\qquad\qquad\quad (\exists\, d)[\text{modDataOf}\,(op, dt, d) \lor \text{accessDataOf}\,(op, dt, d)]\,)\,]\\ &\qquad \supset \quad (\nexists P)\,[\text{inForm}\,(P, \mathcal{F}_1) \land \text{inForm}\,(P\,|^{e}_{dt}, \mathcal{F}_2)]\,]\end{aligned}$$

This constraint is checked by PegaSys whenever a data abstraction is replaced.

6.2. Interaction Refinement

The refinement of interactions (relationships among active entities) must obey the following general procedure. For a relation R of a form $\mathcal{F}_1$, let $\mathcal{F}_2$ denote the form obtained by replacing the relation R by its refinement (a set of one or more relations). What must be shown is that the new form $\mathcal{F}_2$ logically implies the replaced relation R. That is, we require that an interaction be a logical consequence of its more primitive refinement (plus any other relations in $\mathcal{F}_2$). This proof will use the definition of R, and possibly definitions and acceptability constraints of other derived relations. Such proofs are easy, since predicates range over small finite sets and usually need to be evaluated over only one element of a set.

Interaction refinement is illustrated by returning to Figures 2b and 2c of our scenario. In Figure 2b, we have the relation

$$\text{DataFlow}\,(Source, NL\_Sender, msg)$$

which is replaced in Figure 2c by

$$\text{vDataFlow}\,(Source, NL\_Sender, msg) \quad \wedge \quad \text{accessDataOf}\,(NL\_Sender, Source, msg)\ .$$

These relations are not displayed in the figures, but are contained in the complete forms in Appendix A.[5] We must show that $DataFlow(Source, NL\_Sender, msg)$ follows from the entire form for Figure 2c. First note that, by the definition of $vDataFlow$, we have

$$\text{vDataFlow}\,(Source, NL\_Sender, msg) \quad \supset \quad \text{signal}\,(Source, NL\_Sender)\ .$$

Using $signal(Source, NL\_Sender)$ and $accessDataOf(NL\_Sender, Source, msg)$, we get $DataFlow(Source, NL\_Sender, msg)$ (by the definition of $DataFlow$).

7. Conclusions and Future Work

Visual representation of system properties appears to be a highly promising approach to the development, documentation, and maintenance of large software systems. Past experience has shown that humans find it easy to express and communicate certain knowledge about programs graphically.

PegaSys combines the use of graphics with formal logic. Through a coupling of graphics and logical representation, pictures intended to describe the form of a system are given underlying meaning. Thus, PegaSys

[5]Recall that the "kind" properties on arcs in the scenario figures have been suppressed.

is able to support the construction and refinement of system specifications in a way that is not only pictorial (and intuitive), but computationally meaningful.

We feel that PegaSys makes a contribution to the field of visual specification in several ways. First of all, we have found our formulation of the form calculus, the primitive relations we have chosen, and our technique for building derived relations, to be a simple, useful, and powerful system for building a broad class of specifications. We have found it possible to model the structure of not only von Neumann-style systems, but dataflow systems as well. Because of the simplicity of our representation, we have been able to define a general refinement methodology that can be checked automatically. This methodology can also be extended to accomodate specialized restrictions on how derived concepts may be refined.

PegaSys becomes even more interesting when viewed as a complete framework for system development and testing. Future plans for PegaSys include two main objectives.

- A mechanism for connecting a picture hierarchy to actual system code and verifying that the form specified by a picture matches the form of the code. This requires, among other things, a procedure for automatically deriving a form from a program.
- A visual debugging facility, which includes an animator for illustrating the execution of an actual program in the visual framework constructed by the user. Note that our approach to animation alleviates the problem of presenting a mass of intricate computational detail by allowing a user to choose the most beneficial way of viewing system execution. We also plan to incorporate a testing facility for associating predicates with certain icons in pictures and evaluating them during program execution.

Our current research is continuing our focus on the static aspects of PegaSys, which provide a basis for the capabilities mentioned above. Our primary efforts involve the development of an automatic form generator for Ada programs and further work on specification refinement. This includes refinement of both passive and active entities, as well as changes to specifications that constitute a restructuring or reformulation, rather than direct substitution. Work on the dynamic aspects of PegaSys is expected to start in the near term.

REFERENCES

[1] Davis, A.L., and Keller, R.M., Data flow program graphs, *Computer*, vol. 15, no. 2, February 1982, pp. 26-41.

[2] Dennis, J.B., First version of a data flow procedure language, *Lecture Notes in Computer Science*, Springer-Verlag, 1974, pp. 362-376.

[3] Keller, R.M., Jayaraman, B., Rose, D., Lindstrom, G., FGL (Function Graph Language) programmers' guide, Technical report AMPS no. 1, University of Utah, Computer Science Department, July 1980.

[4] Maguire, G.Q., Jr., A graphical workstation and programming environment for data-driven computation. PhD thesis, Department of Computer Science, The University of Utah, March 1983.

[5] Rich, C., and Shrobe, H., Initial report on a Lisp Programmer's Apprentice. *IEEE Transactions on Software Engineering*, vol. SE-4, no. 6, November 1978, pp. 456-466.

[6] Yourdan, E., and Constantine, L.L., *Structured design: Fundamentals of a discipline of computer program and systems design*, Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632, 1979.

Appendix A: Levels and Forms for the Scenario

The following presents the forms for each of the four levels ir ur scenario, which are contained in Figure 7. (Note that Figure 2 contained views of these levels.)

```
Level 1  (Network Service)

         process(Source)
         process(Network_Layer)
         process(Destination)
         type(msg)

         DataFlow(Source,Network_Layer,msg)
         DataFlow(Network_Layer,Destination,msg)


Level 2  (Data Link Service)

         process(Source)
         process(NL_Sender)
         process(Data_Link_Layer)
         process(NL_Receiver)
         process(Destination)
         type(msg)

         DataFlow(Source,NL_Sender,msg)
         DataFlow(NL_Sender,Data_Link_Layer,msg)
         DataFlow(Data_Link_Layer,NL_Receiver,msg)
         DataFlow(NL_Receiver,Destination,msg)


Level 3  (Data Link Architecture)

         process(Source)
```

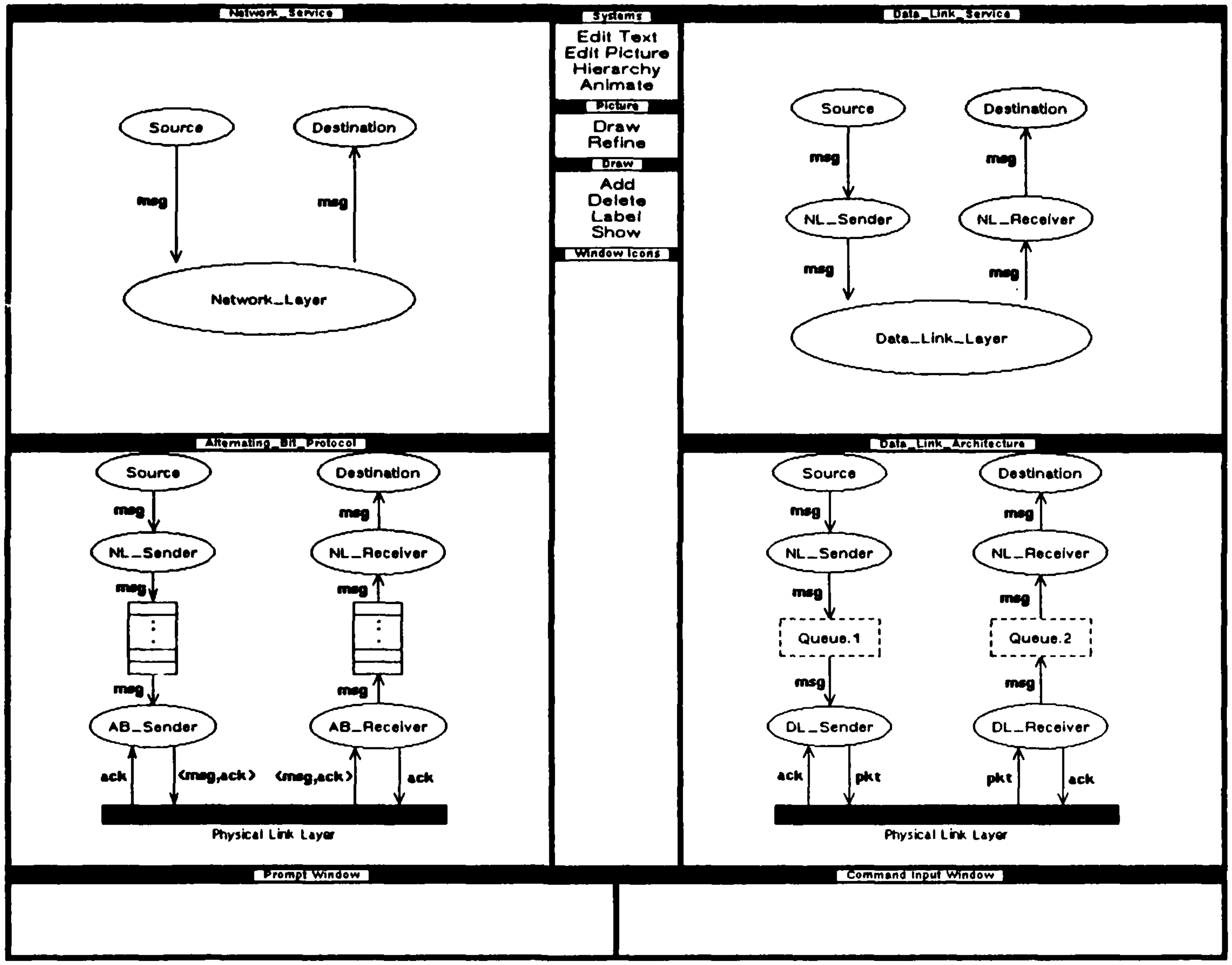


Figure 7: The four levels in the scenario, beginning in the upper-left window and progressing clockwise.

```
process(NL_Sender)
dataType(Queue.1)
process(DL_Sender)
dataType(Physical_Link_Layer)
process(DL_Receiver)
dataType(Queue.2)
process(NL_Receiver)
process(Destination)
type(msg)
type(pkt)
type(ack)
```

Unique internal names Queue.1 and Queue.2 were created to distinguish between two instances of a queue type data abstraction.

```
vDataFlow(Source,NL_Sender,msg)
accessDataOf(NL_Sender,Source,msg)
Write(NL_Sender,Queue.1,msg)
Read(DL_Sender,Queue.1,msg)
Write(DL_Sender,Physical_Link_Layer,pkt)
Read(DL_Sender,Physical_Link_Layer,ack)
Read(DL_Receiver,Physical_Link_Layer,pkt)
Write(DL_Receiver,Physical_Link_Layer,ack)
Write(DL_Receiver,Queue.2,msg)
Read(NL_Receiver,Queue.2,msg)
vDataFlow(NL_Receiver,Destination,msg)
accessDataOf(Destination.NL_Receiver,msg)
```

Notice that `Write(NL_Sender,Queue.1,msg)` and `Read(NL_Receiver,Queue.2,msg)` are not legal refinements of `DataFlow(NL_Sender,Data_Link_Layer,msg)` and `DataFlow(Data_Link_Layer,NL_Receiver,msg)` according to the methodology explained in this paper. This is a simple instance of a more more complex type of refinement presently under investigation. Note however, that `Write(NL_Sender,Queue.1,msg)` intuitively implies some signal and data transfer between `NL_Sender` and an operation of abstraction `Queue.1`. In fact, this refinement is made in the next layer.

In addition, note that this level makes a complete transition from the dataflow to the von Neumann model.

Level 4 (Alternating Bit Protocol)

```
        process(Source)
        process(NL_Sender)
        process(AB_Sender)
        dataType(Physical_Link_Layer)
        process(AB_Receiver)
        process(NL_Receiver)
        process(Destination)
        type(msg)
        type(ack)
        tuple(<msg,ack>)

        vDataFlow(Source,NL_Sender,msg)
        accessDataOf(NL_Sender,Source,msg)
        Signal(NL_Sender,Enq.1)
        accessDataOf(Enq.1,NL_Sender,msg)
        Write(NL_Sender,Enq.1,msg)
        Read(AB_Sender,Deq.1,msg)
        Write(AB_Sender,Physical_Link_Layer,<msg,ack>)
        Read(AB_Sender,Physical_Link_Layer,ack)
        Read(AB_Receiver,Physical_Link_Layer,<msg,ack>)
        Write(AB_Receiver,Physical_Link_Layer,ack)
        Write(AB_Receiver,Enq.2,msg)
        Read(NL_Receiver,Deq.2,msg)
        Signal(Deq.2,NL_Receiver)
        vDataFlow(NL_Receiver,Destination,msg)
        accessDataOf(Destination,NL_Receiver,msg)
```

The following is added as a refinement of Queue.1 and Queue.2.

```
        dataType(Asyn_FIFO_Queue.1)
        process(Enq.1)
        process(Deq.1)
        modDataOf(Enq.1,Asyn_FIFO_Queue.1,epsilon)
        accessDataOf(Deq.1,Asyn_FIFO_Queue.1,epsilon)

        dataType(Asyn_FIFO_Queue.2)
```

```
process(Enq.2)
process(Deq.2)
modDataOf(Enq.2,Asyn_FIFO_Queue.2,epsilon)
accessDataOf(Deq.2,Asyn_FIFO_Queue.2,epsilon)
```

Appendix B: More Derived Relations

This appendix presents several examples of derived relations, none of which appear in the scenario. The networking example dealt with processes, data abstractions, and values; the relations discussed below deal with subprograms and names. Figure 8 contains seven derived relations for use in von Neumann-style specifications; four deal with subprogram calls and two with side effects. As before, associated with each derived relation is its acceptability constraint and definition.

Five calling relations are defined in Figure 8. A parameterless subprogram call in which no data is communicated, is defined by B1. Three subprogram calls, each of which differs in its method of data communication, are defined by B2-B4. In line with the philosophy behind the form calculus, these relations do not dictate how specified data communication is to be implemented. It can be done by means of explicit parameter passing or through global shared variables, whichever is appropriate.

The relation *CallByValue* specifies that values are transmitted from x to y, while *ReturnValue* specifies that a value is transmitted back to y from x. A combination of these relations would be used to specify a subprogram call having both passed and returned values. Call by reference, at B4, differs in that names, not values, are transmitted. Finally, at B5, a generic subprogram call is defined to be any of the four possibilities B1-B4.

Two notions of side effects are defined, both of which are concerned with the modification of data. A simple notion of a side effect is defined at B6, which says that x has a side effect on y if x modifies y's data and $x \neq y$. A more subtle notion is defined at B7, which describes side effects that result because of aliasing. It says that x may have a side effect on y

because x modifies a data object referenced by a name aliased to a name owned by y. The predicate *contained*(n, x) is defined to be

$$accessDataOf(x, x, n.val) \quad \vee \quad modDataOf(x, x, n)$$

and is used to model the fact that n "belongs to" x. Observe that *SideEffectThroughAliasing* will still be satisfied if z is the same as x, i.e., n_1 may be declared in x.

B1. SimpleCall (x, y) :

- operation $(x) \wedge$ subprogram (y)
- control $(x, y) \wedge$ returnOfControl (y, x)

B2. CallByValue (x, y, v) :

- operation $(x) \wedge$ subprogram $(y) \wedge$ value (v)
- control $(x, y) \wedge$ returnOfControl $(y, x) \wedge$ accessDataOf (y, x, v)

B3. ReturnValue (x, y, v) :

- subprogram $(x) \wedge$ operation $(y) \wedge$ value (v)
- control $(y, x) \wedge$ returnOfControl $(x, y) \wedge$ accessDataOf (y, x, v)

B4. CallByRef (x, y, n) :

- operation $(x) \wedge$ subprogram $(y) \wedge$ name (n)
- control $(x, y) \wedge$ returnOfControl $(y, x) \wedge$ modDataOf (y, x, n)

B5. Call (x, y, n, v_1, v_2) :

- operation $(x) \wedge$ subprogram $(y) \wedge$ name $(n) \wedge$ value $(v_1) \wedge$ value (v_2)
- $n \neq \epsilon \supset$ CallByRef $(x, y, n) \quad \wedge$
  $v_1 \neq \epsilon \supset$ CallByValue $(x, y, v_1) \quad \wedge$
  $v_2 \neq \epsilon \supset$ ReturnValue $(y, x, v_2) \quad \wedge$
  $v_1 = v_2 = n = \epsilon \supset$ SimpleCall (x, y)

B6. SideEffect (x, y, n) :

- subprogram $(x) \wedge$ operation $(y) \wedge$ name (n)
- modDataOf $(x, y, n) \wedge x \neq y$

B7. SideEffectThroughAliasing (x, y, n_1, n_2) :

- subprogram $(x) \wedge$ operation $(y) \wedge$ name $(n_1) \wedge$ name (n_2)
- $(\exists z)$ [contained $(n_1, z) \wedge$ modDataOf(x, z, n_1)] $\quad \wedge$
  aliased $(n_1, n_2) \wedge$ contained (n_2, y)

Figure 8: Six derived relations for von Neumann specifications.

DISTRIBUTION LIST

addresses	number of copies
Douglas A. White RADC/COES	20
RADC/TSTD GRIFFISS AFB NY 13441	1
RADC/DAP GRIFFISS AFB NY 13441	2
ADMINISTRATOR DEF TECH INF CTR ATTN: DTIC-DDA CAMERON STA BG 5 ALEXANDRIA VA 22314	12
HQ ESC (XPZP) SAN ANTONIO TX 78243	1
HQ ESC/DOO SAN ANTONIO TX 78243	1
DMA HYDROGRAPHIC/TOPOGRAPHIC CENTER ATTN: STT WASHINGTON DC 20315	2
HQ USAF/SAMI WASHINGTON DC 20330	1

HQ USAF/SITT WASHINGTON DC 20330	1
HQ USAF/RDSS WASHINGTON DC 20330	1
DIRECTOR DMAHTC ATTN: SDSIM 6500 Brookes Lane WASH DC 20315	1
RADC/ISISI Bldg 3, Rm 43 Griffiss AFB NY 13441	1
PENTAGON USDR&E, RM 3E-187 ATTN: C3I WASHINGTON DC 20301	2
HQ AFSC/DLAE ANDREWS AFB DC 20334	1
HQ AFSC/SDE ANDREWS AFB DC 20334	1
HQ AFSC/XRKR ANDREWS AFB MD 20334	1
HQ AFSC/XRK ANDREWS AFB MD 20334	1

HQ SAC/NRI (STINFO LIBRARY) OFFUTT AFB NE 68113	1
3246 TESTW/TZE EGLIN AFB FL 32542	1
TAFIG/IIDD LANGLEY AFB VA 23665	1
HQ TAC/XPS (STINFO)] LANGLEY AFB VA 23665	1
MAJOR JOHN MORRISON USMTM/JOINT SECTION APO NEW YORK 09038	2
HQ TAC/DOY LANGLEY AFB VA 23565	1
HQ TAC/DRCC LANGLEY AFB VA 23565	1
AFSC LIAISON OFFICE LANGLEY RESEARCH CENTER (NASA) LANGLEY AFB VA 23665	1
HQ TAC/DOF LANGLEY AFB VA 23565	1

ASD/ENSSA WRIGHT-PATTERSON AFB OH 45433	1
AFWL/SUL ATTN: TECHNICAL LIBRARY KIRTLAND AFB NM 87117	1
ASD/RWEE ATTN: MR LARRY WEAVER WRIGHT-PATTERSON AFB OH 45433	1
ASD/ENEGA WRIGHT-PATTERSON AFB OH 45433	1
ASD/ENAMW WRIGHT-PATTERSON AFB OH 45433	1
ASD/XRS WRIGHT-PATTERSON AFB OH 45433	1
AFIT/LDEE - TECHNICAL LIBRARY BUILDING 640, AREA B WRIGHT-PATTERSON AFB OH 45433	1
AFWAL/MLTE WRIGHT-PATTERSON AFB OH 45433	1
AFAMRL/HE WRIGHT-PATTERSON AFB OH 45433	1

AFHRL/LRS-TDC WRIGHT-PATTERSON AFB OH 45433	1
ASD/EN ATTN: MR JEFFERY L. PESLER, STAFF ENGINEER ASD COMPUTER RESOURCE FOCAL POINT OFFICE WRIGHT-PATTERSON AFB OH 45433	1
ASD/AFALD/AXT WRIGHT-PATTERSON AFB OH 45433	1
AFHRL/OTS Williams AFB AZ 85224	1
AUL/LSE 67-342 MAXWELL AFB AL 36112	1
HQ AFCC/DAPL BLDG P-40 NORTH, RM 9 SCOTT AFB IL 62225	1
AWS Technical Library FL4414 SCOTT AFB IL 62225	1
AFHRL/ID LOWRY AFB CO 80230	1
3420 TCHTG/TTMNL LOWRY AFB CO 80230	1

CODE R141B TECHNICAL LIBRARY DEFENSE COMMUNICATIONS ENGINEERING CENTER 1860 WIEHLE AVENUE RESTON VA 22090	1
COMMAND CONTROL AND COMMUNICATIONS DIV DEVELOPMENT CENTER MARINE CORPS DEVELOPMENT & EDUCATION COMMAND ATTN: CODE D10 QUANTICO VA 22134	1
EDWARD D GRAHAM, JR DIVISION 2101 SANDIA NATIONAL LABORATORIES ALBUQUERQUE NM 87112	1
AFLMC/LGY ATTN: CH, SYS ENGR DIV GUNTER AFS AL 36114	1
COMMANDER BALLISTIC MISSILE DEFENSE SYSTEMS COMMAND ATTN: BMDSC-AOLIB PO BOX 1500 HUNTSVILLE AL 35807	1
DIRECTOR BMD ADVANCED TECHNOLOGY CENTER ATTN: ATC-D, FRANK L BROWN PO BOX 1500 HUNTSVILLE AL 35807	1
DIRECTOR BMD ADVANCED TECHNOLOGY CENTER ATTN: ATC-P, CHARLES VICK PO BOX 1500 HUNTSVILLE AL 35807	1
DET 1, AFOSR EOARD/CI TECHNICAL INFORMATION OFFICE BOX 14 FPO NEW YORK NY 09510	1
COMMANDING OFFICER NAVAL AVIONICS CENTER LIBRARY - CODE 765 INDIANAPOLIS IN 46218	1

COMMANDING OFFICER 1
NAVAL TRAINING EQUIPMENT CENTER
TECHNICAL INFORMATION CENTER
BUILDING 2068
ORLANDO FL 32813

COMMANDER 1
NAVAL OCEAN SYSTEMS CENTER
ATTN: TECHNICAL LIBRARY, CODE 4473B
SAN DIEGO CA 92152

US NAVAL WEAPONS CENTER, CODE 343 1
ATTN: TECHNICAL LIBRARY
CHINA LAKE CA 93555

SUPERINTENDENT (CODE 1424) 1
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93940

COMMANDING OFFICER 1
NAVAL RESEARCH LABORATORY
CODE 2627
WASHINGTON DC 20375

NAVELEXSYCOM 1
PME-117-22
WASHINGTON DC 20360

COMMANDER 1
US ARMY ELECTRONIC WARFARE LABORATORY
OFFICE OF MISSILE ELECTRONIC WARFARE
ATTN: DELEW-M-FM (MR ANDERSON)
WHITE SANDS MISSILE RANGE NM 88002

REDSTONE SCIENTIFIC INFORMATION CENTER 2
ATTN: DRSMI-RPRD
US ARMY MISSILE COMMAND
REDSTONE ARSENAL AL 35809

MILITARY SEALIFT COMMAND 1
TECHNICAL INFORMATION CENTER M-16
DEPARTMENT OF THE NAVY
WASH DC 20390

ADVISORY GROUP ON ELECTRON DEVICES 2
FTS (FEDERAL COMM SYSTEM)
201 VARICK STREET, Rm 1140
NEW YORK NY 10014

FRANK J SEILER RESEARCH LAB 1
FJSRL/NHL
US AIR FORCE ACADEMY CO 80840

LOS ALAMOS SCIENTIFIC LABORATORY 1
ATTN: REPORT LIBRARY
MAIL STATION 5000
PO BOX 1663
LOS ALAMOS NM 87545

AIR FORCE ELEMENT (AFELM) 1
THE RAND CORP
1700 MAIN STREET
SANTA MONICA CA 90406

AEDC LIBRARY (TECH FILES) 1
ARNOLD AFS TN 37389

Director 1
National Security Agency
ATTN: T5112/TDL
Fort Meade MD 20755

Director 1
National Security Agency
ATTN: W22
Fort Meade MD 20755

Director 1
National Security Agency
ATTN: W31
Fort Meade MD 20755

1
Director
National Security Agency
Attn: R-8314 (Mr. Alley)
Fort Meade MD 20755

Director National Security Agency ATTN: S63 Fort Meade MD 20755	1
Director National Security Agency ATTN: S07 Fort Meade MD 20755	1
Director National Security Agency ATTN: R21 VICE R2 Fort Meade MD 20755	1
Director National Security Agency ATTN: R5 Fort Meade MD 20755	1
Director National Security Agency ATTN: R02(T) (Mr. Orlosky) Fort Meade MD 20755	1
Director National Security Agency ATTN: R7 Fort Meade MD 20755	1
Director National Security Agency ATTN: R8 Fort Meade MD 20755	1
Director National Security Agency ATTN: P207 Fort Meade MD 20755	1
HQ ESD/FASE HANSCOM AFB MA 01731	1

HQ ESD/TCSR (BURT HOPKINS) HANSCOM AFB MA 01731	1
HQ ESD/TCIE HANSCOM AFB MA 01731	1
ESD/XRT HANSCOM AFB MA 01731	1
ESD/XRVT HANSCOM AFB MA 01731	1
ESD/XRW HANSCOM AFB MA 01731	1
ESD/XRTR HANSCOM AFB MA 01731	1
ESD/TCG (MR RON LANZA) HANSCOM AFB MA 01731	1
ESD/XRC (AFSC) HANSCOM AFB MA 01731	1
ESD/XR HANSCOM AFB MA 01731	2

HQ ESD/DCR-1I 1
HANSCOM AFB MA 01731

AFEWC/ESRI 1
San Antonio TX 78243

485 EIG/EIEXR (DMO) 2
Griffiss AFB NY 13441

ESD/TCIE 1
ATTN: MR CLIFTON DOIRON
HANSCOM AFB MA 01731

ESD/TCS-2 1
ATTN: C A MATHEWSON
HANSCOM AFB MA 01731

ESD/TCS-1D 1
ATTN: LT ROBERT GINGRICH
HANSCOM AFB MA 01731

ESD/TCS-1D 1
ATTN: LT JEANNE MURTAGH
HANSCOM AFB MA 01731

ESD/TCS-1D 1
ATTN: LT TERRY TAYLOR
HANSCOM AFB MA 01731

ESD/TCS-1D 1
ATTN: E D SPRAGUE
HANSCOM AFB MA 01731

NAVELEX DET PAX 1
PME 120-132

NAVAL AIR STATION
PATUXENT RIVER MD 20670

ASD/AXPP 1
WRIGHT-PATTERSON AFB OH 45433

ASD/AXPM 1
WRIGHT-PATTERSON AFB OH 45433

ASD-AFALD/AXAE 1
WRIGHT-PATTERSON AFB OH 45433

ASD-AFALD/AXT 1
WRIGHT-PATTERSON AFB OH 45433

1839 EIG/EIEM 1
KEESLER AFB MS 39534

Dr. Mark S. Moriconi 5
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025-3493

LCDR Ronald B. Ohlander 1
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209-2389

Dr. Craig I. Fields Sys. Sciences Div., Def. Sciences Office Defense Advanced Research Projects Agency 1430 Wilson Boulevard Arlington, VA 22209-2389	1
Dr. David Fox, Director Mathematical & Information Sciences AFOSR/NM Bldg. 410, Bolling AFB Washington, DC 20332	1
Dr. Robert B. Grafton Office of Naval Research Code 433 800 N.Quincy St. Arlington, VA 22217	1
Dr. Glen Allgaier Naval Ocean Systems Center Code 8242 271 Catalina Boulevard San Diego, CA 92152	1
Dr. Robert M. Balzer University of Southern Cal. Information Sciences Institute 4676 Admiralty Way Marina Del Ray, CA 90291	1
Dr. Cordell Green Kestrel Institute 1801 Page Mill Road Palo Alto, CA 94304	1
Mr. Thomas E. Cheatham Harvard University Aiken Computation Laboratory 33 Oxford Street Cambridge, MA 02138	1
Dr. David C. Luckham Stanford University Computer Systems Laboratory Stanford, CA 94305	1

Dr. Charles Rich AI Laboratory MIT (NE43-350) 545 Technology Square Cambridge, MA 02139	1
Ms Lorraine M. Duvall Director of Research IIT Research Institute 199 Liberty Plaza Rome, NY 13440	1
Dr. Elaine Kant Carnegie-Mellon University Computer Science Department Schenley Park Pittsburg, PA 15213	1
Dr. Bernard A. Kulp Chief Scientist AFSC/DLZ Andrews AFB Washington, DC 20334	1
Dr. Karl N. Levitt SRI International Computer Science Laboratory 333 Ravenswood Ave. Menlo Park, CA 94025	1
Dr. Richard Henry Brown MITRE Corp. P.O. Box 208 Bedford, MA 01730	1
Dr. Brian McCune Advanced Information & Decision Systems Suite 224 201 San Antonia Circle Mountin View, CA 94040	1
Mr. John Entzminger Director DARPA/TTO 1400 Wilson Blvd. Arlington, VA 22209-2389	1
Dr. Stephen Squires Defense Advanced Projects Agency Information Processing Techniques Office 1400 Wilson Blvd Arlington, VA 22209	1