University of Waterloo
Department of Computer Science
Waterloo, Ontario, Canada

Technical Report Series

CS-93-52

Abstract Data Views: A Module Interconnection Concept to Enhance Design for Reusability

by

D.D. Cowan C.J.P. Lucena

November, 1993

Abstract Data Views: A Module Interconnection Concept to Enhance Design for Reusability

D.D. Cowan
Computer Science Department & Computer Systems Group
University of Waterloo
Waterloo Ontario,
Canada
N2L 3G1
dcowan@csg.uwaterloo.ca

C.J.P. Lucena
Departamento de Informática
Pontifícia Universidade Católica
Rio de Janeiro, 22453-900, RJ,
Brazil
lucena@inf.puc-rio.br

November 7, 1993

Abstract

The Abstract Data View (ADV) design model was originally created to specify clearly and formally the separation of the user interface from the application component or Abstract Data Type (ADT), and to provide a systematic design method that is independent of specific application environments. Such a method should lead to a high degree of reuse of both interface components and their associated ADTs. The material in this paper extends the concept of ADVs to encompass the general specification of interfaces between objects in the same or different computing environments. This approach to specifying interfaces clearly separates objects from each other, since objects do not need to know how they are used, or how they obtain services from other objects. Thus, objects which are designed to minimize knowledge of the environment in which they are used, are more amenable to reuse.

Categories and Subject Descriptors: D.1.5 [**Software**]: Programming Techniques – *Object-oriented Programming*; D.2.2 [**Software**]: Software Engineering – *Tools and Techniques*; D.2.6 [**Software**]: Software Engineering – *Programming Environments* D.2.10 [**Software**]: Software Engineering – *Design*; D.2.m [**Software**]: Software Engineering – *Miscellaneous*; General Terms: Abstract Data Types, Interactive Applications, Programming, User Interfaces

Additional Key Words and Phrases: End-User Programming, Script Languages, Interfaces

1 Introduction

The concept of Abstract Data View (ADV) [CILS93a, CILS93b] is intended to bridge the gap between the internal world of the application object or abstract data type (ADT) and its requirement for knowledge of the external world. An ADT should not know how its state is presented to the user, or how the user interacts with that presentation. In addition, knowledge of the external world should be provided in such a way that it can be disconnected from the application object when it is no longer needed. For example, an integer which is part of the state of an ADT, could be presented as a numeric quantity, a position on a dial, or the position of a slider, and the user could interact with the interface to change the integer in several different ways. As well, the ADT should not know which representation is used. ADVs support this concept of a clear separation of external representation from internal state.

ADVs as user interfaces disconnect the application object from the user; the application object has no knowledge of the interaction with the user. All interaction is encapsulated in the ADV which is then provided to the object through its provided services. ADVs can also be composed from other ADVs through nesting; thus, it is possible to build complex user interfaces from simpler components. An ADV is an object or ADT in that it has a state and an interface which can be operated by an external agent.

The concept of ADV was first applied to man-machine interfaces where a number of different approaches such as those reported in [KP88, Hil92, CCCL93] have separated the interface from the application in an attempt to reuse interface objects. The original ADV approach allowed both the association of different interfaces to the same server and the composition of interfaces through nesting [CILS93a, CILS93b], thus, providing for the reuse of interfaces and their associated ADTs. Extensive experimentation with the ADV concept for the design of man-machine interfaces has led us to believe that the concept can be generalized and extended to model internal module or object interconnection interfaces and interfaces to other media such as networks. ADVs have been used to support user interfaces for games and a graph editor [C$^+$92], to interconnect modules in a user interface design system (UIDS) [LCP92], to support concurrency in a cooperative drawing tool and to implement a ray tracer in a distributed environment. The VX·REXX [VRe93] system that was built as a research prototype was motivated by the idea of composing applications in the ADV/ADT style.

ADVs can be used to connect objects together at different binding times during the programming process, that is at code-time, link-time or run-time. Also ADVs can be used to connect together small and large objects, that is, they encompass programming-in-the-small and programming-in-the-large [DK76]. Currently ADVs are a design concept and can be implemented using many different strategies. For example, VX·REXX supports dynamic programming-in-the-large and allows application integration by linking together user interfaces (ADVs) and large predefined objects.

We believe it should be possible to balance decomposition by form and decomposition by function [Mah90] in the software design process in a manner similar to the way it is practiced in various engineering disciplines. In order to achieve this goal we need both design and implementation tools to merge the structured and object-oriented design approaches. We believe the concept of Abstract Data Views provides the bridge between these two design approaches, because it supports both the object view and decomposition by nesting.

2 Some Design Issues

An argument for the justification of contemporary object-oriented design methodologies, that is often used, emphasizes that object-oriented design and development inverts the traditional function-oriented methodologies often associated with structured analysis and design [DeM78]. With this inversion the emphasis is no longer placed on specifying and decomposing system functionality, rather the object-oriented approach focuses first on identifying objects from the application domain, and then associating procedures with them. A claim, that often follows from the previous argument, is that object-oriented development not only allows information to be shared within an application, but also offers the prospect for reuse of both designs and code. We believe that this argument is essentially incomplete, provides only a partial view of the very complex software design problem, and that software design requires both approaches.

A comprehensive design approach should at least include both function-oriented and object-oriented (form) techniques. In order to provide a perspective on this problem we first briefly review some concepts from design theory followed by a brief justification of the previous statement.

There exist four widely known design paradigms (abstract prescriptive models of the design process) in Computer Science [Das91]: the analysis-synthesis-evaluation paradigm, the artificial intelligence paradigm (design problems are said to be ill-structured), the algorithmic paradigm (design problems are considered well-structured problems - designs can be "programmed" from precise requirements) and the formal design paradigm (design-while-verify).

The classical analysis-synthesis-evaluation paradigm (ASE) [Ale64] represents a widely held view of the design process in most engineering disciplines. According to ASE, the design process is thought to consist of three logically and temporally distinct stages: a stage of analysis of the requirements, followed by a stage of synthesis, followed by a stage of evaluation. In general, several instances of this three stage sequence may be required in order to progress from a more abstract to a more concrete level. Regardless of whether one considers large-scale software development or the design of small programs, the development of software design methods has been enormously influenced by the ASE paradigm.

Several recent models of the software development process such as prototyping [B$^+$92] have overcome the limitations of the classical ASE paradigm by viewing it as an interweaving of solutions (designs or forms), problems (requirements), and evaluations, in which solutions and evaluations prompt problems as much as problems prompt solutions.

Maher [Mah90] has categorized the most important design synthesis models associated with the new form of the ASE design process. This process is often presented in three interwoven stages called problem statement, design synthesis of a candidate solution and evaluation. These design synthesis models are decomposition, reasoning by cases and transformation. Decomposition is the classical "divide-and-conquer" approach to design and is the most widely used synthesis technique in software development. Reasoning by cases is the use of reasoning by analogy and is represented by recent approaches to design reuse [Mai91] and transformational synthesis includes various forms of program derivation from formal specifications [Par90].

Let us concentrate on decomposition because of its relevance to practical software development. There is both decomposition by function and decomposition by form. Maher [Mah90] calls attention to the classical "function versus form" dilemma. Engineers, architects and other designers often face this problem: how much of each kind of decomposition to use. In the next paragraph we

paraphrase some material from Maher [Mah90] in order to explain these two design strategies.

What is decomposition by form and decomposition by function? First a design problem is decomposed into subproblems and decomposition usually follows one of these two approaches. In the approach called decomposition by form one divides a problem in a domain of design knowledge into the various components. For example, in structural design these components would be the objects such as walls, slabs, and steel beams that are used to construct the design solution. In decomposition by function one divides the solution into the various functions that must be provided by the design solution. For example, a structural solution must resist certain loads, provide unobstructed open space, and have certain insulating properties. Decomposition by form is an object-centred approach in which the design knowledge is centred around the physical components. Decomposition by function considers a functional decomposition to a sufficient level of detail so that a function to form mapping can occur.

Until very recently software engineers have been constrained by a lack of tools and hence have tended to use only functional decomposition. Although this is an appropriate problem-solving approach, decomposition involves re-composition later to produce the final design. This design obtained through re-composition may contain several problems. For instance, if the requirements of the design problem change, a system based on decomposing functionality may require massive restructuring.

Object-oriented design has allowed the software engineer to perform decomposition by form. Unfortunately, much of the literature in object-oriented development assumes that decomposition by form (object) replaces decomposition by function, and further, that inheritance is the way decomposition by function takes place in object-oriented design. In fact, inheritance allows specialization of forms, it is not meant to support decomposition by function into different smaller units.

We believe that we are now able to balance decomposition by form and decomposition by function in the software design process in a manner similar to the way it is practiced in various engineering disciplines. In order to achieve this goal we need both design and implementation tools to merge the structured and object-oriented design approaches. We believe the concept of Abstract Data Views (ADVs) with some modifications to the concept of Abstract Data Types (ADTs) provides the bridge between these two design approaches.

3 Abstract Data Types and Abstract Data Views

3.1 Abstract Data Types

An abstract data type (ADT) is an object in the object-oriented sense of program design. We use the term ADT because we are mostly interested in its properties as a type. An ADT is an object which is defined by a state and a functional interface. The functions defined by the interface can query or change the state of the ADT. Only the functional interface is accessible to external sources. Given an ADT X we can instantiate various objects of type X.

We extend the concept of ADT to allow nesting of ADTs; nesting implies that ADTs are composed of a number of constituent objects and that one ADT encloses its constituents. Also the enclosing ADT knows the identity of its constituents, but the enclosed objects do not know the identity of the enclosing object. The semantics of nesting are such that reference to enclosed

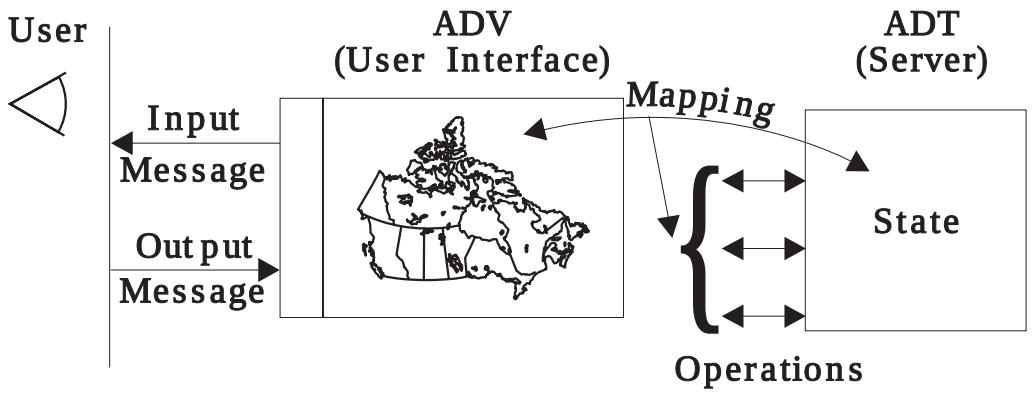

Figure 1: An ADV as a user interface

objects from outside the enclosing object do not violate encapsulation [CL93]. Nesting supports decomposition by function.

3.2 Abstract Data Views as User Interfaces

Abstract data views (ADVs) are ADTs which have been modified to support the design of user and module interfaces. ADVs were originally conceived to act as general user interfaces and to achieve a separation of concerns by providing a clear separation at the design level between the application as represented by an ADT, and the user interface or ADV.

ADVs are objects in that they have a state, and a functional interface; they have also been extended to support nesting. ADVs support one or more mappings which allow the ADV to query the state of any associated ADT and to change the state of that ADT through its functional interface. This mapping approach allows controlled access to the state of the ADT, thus preserving the hiding principle [Par72b]. The strategy used to implement this mapping is not specified in the model.

The functional interface of an ADV is not invoked through the usual procedure or function calls but by input messages. Input messages can be triggered by external operations such as input events caused by a keyboard or a mouse. ADVs can produce output messages that use the mapping to query the values of the variables in an accompanying ADT. An output message is sent every time the ADT operations change the state of the ADT. Output messages in terms of a user interface are the display commands which paint various views on the screen. Of course output messages do not change the state of the associated ADT. Figure 1 illustrates the concepts described in the preceding paragraphs.

In the context of user interfaces and other transformations from one medium to another the input and output messages can be viewed as medium transformers. For example, the input message or event takes a user action and transforms that action into a sequence of ADV operations. The output message or display takes a sequence of ADV operations and transforms them into a viewable or tactile phenomenon.

Type *ADT_Name*

Declaration: *var_name*1: *Type*1, *var_name*2: *Type*2, ...

Invariant: *inv-name* $\triangleq$ *formulae*

Type *ADT_Name*1

Declaration: *var_name*11: *Type*1, *var_name*12: *Type*2, ...

Invariant: *inv-name* $\triangleq$ *formulae*

Function *function_name*1 (*input_variables*) *res*: *output_variables*
external wr *var_name*1, wr *varname*12, rd *var_name*2
pre-condition: *formulae*
post-condition: *formulae*
⋮

End *ADT_Name*1

Function *function_name* (*input_variables*) *res*: *output_variables*
external wr *var_name*1, rd *var_name*2
pre-condition: *formulae*
post-condition: *formulae*
⋮

End *ADT_Name*

Figure 2: An Abstract Data Type (ADT) Specification Schema

3.3 Specification Schemas for ADTs and ADVs

In order to provide a specification schema for ADTs and ADVs we use a model-based approach to specification based on a VDM-like notation [Ier91]. VDM is used as the notation in which the variable abstract types, the invariant formulae and the pre- and post-conditions are expressed. Since the pre- and post-conditions may refer to the values of the variables before and after the execution of the event, we use VDM hooked variables (e.g. $\overleftarrow{x}$) to denote the variables before the execution.

An abstract schema for our form of ADT is shown in Figure 2. The schema shows *ADT_Name*1 enclosed or nested inside *ADT_Name*. The ADT labelled *ADT_Name* is aware that it is composed of ADTs such as *ADT_Name*1, but *ADT_Name*1 is not aware of any aspect of the state of *ADT_Name*. Variables that are declared "external *varnames*" in a function definition are accessible outside the ADT. All functions are public unless explicitly labelled private. Thus, this form of composition using nesting does not violate encapsulation and enclosed ADTs such as *ADT_Name*1 are independent of their enclosing ADTs.

Although this nesting notation extends VDM, this concept can be mapped back to "standard" VDM notation. The main idea is to use VDM *maps* [Ier91] to keep enclosing ADTs such as *ADT_Name* aware of the nested objects such as *ADT_Name*1.

ADV *name*[**For Type** *adt_names*]

Declaration: *var_name*1: *Type*1, *var_name*2: *Type*2, ...

Invariant: *inv-name* $\triangleq$ *formulae*

ADV *name*1[**For Type** *adt_names*1]

⋮

End *name*1

MESSAGE *Input*1 (*parms*)
external *varnames*
post-condition: *formulae*

MESSAGE *Input*2 (*parms*)

⋮

MESSAGE *Output* ()

⋮

End *name*

Figure 3: An Abstract Data View (ADV) Specification Schema for a user interface

An abstract schema of an ADV for a user interface is illustrated in Figure 3. The ADV specification schema in Figure 3 shows a nesting of ADVs, each one of them possibly referring to an ADT; in this case, the ADV labelled *name* encapsulates the ADV *name*1. Each ADV defines its state through variable declarations whose relationship is expressed by a state invariant. In the ADV labelled *name* the expression "Invariant: *inv-name* $\triangleq$ *formulae*" is the state invariant. The examples in this paper will ignore the invariant property of an ADV or ADT, since we are concentrating on programming aspects of the design approach to interfaces and not with providing exact formal specifications.

When comparing the notation used with the ADV in Figure 3 with VDM, the reader may notice the absence of preconditions. That simply indicates that the programming approach does not assume any restriction on user operations.

External definitions such as "external *varnames*" make external variables accessible internally. The declaration "post-condition: *formulae*" states the post-conditions for a message and represents a logical definition that program code for the message must satisfy.

The ADV approach assumes that the application ADTs never include operations related to user interface aspects of the application. In our approach, the ADV represents all user interface aspects of ADTs. In the syntax of the schema, the representation relation requires that the ADT name be given after the key words **For Type** in the ADV heading. Whenever the ADV has no associated ADT, that is, when it represents only interface concepts independent of the applications such as scrolling, the **For Type** declaration is omitted.

Whenever the declaration **For Type** is used, the pseudo-variable *owner* associated with an

instance of the declared ADT is made available inside the ADV. This pseudo-variable refers to the represented ADT and allows controlled access to the state of the ADT. An ADT can only be modified through its operations but read-only access to its state can be provided through the pseudo-variable *owner*. The *owner* pseudo-variable can then be used in the state invariant for the ADV and, in a restricted form, in the post-conditions, thus, allowing the "appearance" of an ADV to conform to the state of an ADT. This connection between the ADT and ADV captures the WYSIWYG nature of a user interface.

In the original ADV paper [CILS93a] we provided a formal semantics for ADVs as user interfaces, whereas in this paper we illustrate through informal arguments and examples, how ADVs may be used as module interfaces in general program design. In another paper [CCL93] we present the ADV concepts using a visual formalism, and also outline how these concepts may be incorporated in a design methodology to be supported by an appropriate environment.

3.4 Communication Between ADVs and ADTs

Communication between an ADV and an ADT occurs when the ADV executes synchronous invocations (procedure calls, effectively) to the ADT [CCCL93]. An ADT never generates events that must be handled asynchronously by the associated ADV. This approach contrasts with other user interface models, where a component must handle both synchronous and asynchronous invocations from other components. Handling asynchronous invocations is considerably more complicated than handling synchronous invocations since it requires error-prone mechanisms such as signals, interrupts, or callbacks. With an explicit mapping we enforce a one-way communication, and, as a consequence, have fewer interconnections, thus, ensuring that the role and scope of the interface are defined unambiguously. Asynchrony is needed in other models because the "official" locus of control is in the non-user-interface application. This approach is consistent with "slightly-interactive" programs, which mostly compute but occasionally prompt the user for input. In highly interactive applications, however, the actual locus of control is associated with the user. The tension between these two loci of control is the source of the complexity. The ADV model avoids this tension by placing the main locus of control in the ADV. Fundamentally, the ADV model is based on the program waiting for the user rather than the user waiting for the program.

3.5 ADVs and Metaphors

Because of their general properties ADVs can be viewed at the design level as metaphors transforming between the semantic domain of the user and an associated ADT or between the semantic domains of two ADTs. This concept of ADV as a metaphor allows ADVs to be used as user interfaces or to define interfaces between ADTs in the same or separate computers. In this latter context ADVs can be viewed as module interconnection relations. In the following sections we will examine ADVs in all these uses.

A conceptual architecture has been proposed in [BKL93] in which a metaphor level is viewed as the conceptual foundation on which an interactive system is based. Furthermore, the authors claim that the basis of the metaphor level is an ADT.

We have arrived independently at a similar conclusion in that we view the ADV (a special type of ADT) as a metaphor. An example of the nature of an ADV as a metaphor is presented in a later

section of this paper. ADVs as metaphors can also be supported by the central ideas of a formal theory of metaphors [Ind87] which is described in the following paragraphs.

To provide a simplified version of Indurkhya's formal definition of metaphors, we need first to consider V to be a set of symbols and f: $V \rightarrow T$ a function which maps the symbols to a set of types. A vocabulary is defined as a pair $< V, f >$. We now consider a set of well-formed sentences S over the vocabulary $< V, f >$ and a sub-set S_d of S composed of derivations (sentences that define or assign some meaning to the symbols of V).

We can now define a domain D_i as a four-tuple $< V, f, S_d, S >$. The set of structural constraints S delimits the interpretations that can be given to the symbols defined in S_d and to the sentences that can be formed using the vocabulary $< V, f >$. We can now define a metaphor in Indurkhya's style: given two domains $D_1 = < V_1, f_1, S_{d1}, S_1 >$, called the source domain, and $D_2 = < V_2, f_2, S_{d2}, S_2 >$, called the target domain, and an admissible mapping m: $V_1 \rightarrow V_2$, a metaphor is a pair $< m, S >$ where S is a sub-set of S_1 formed by its transformable sentences.

ADVs as models of metaphors adhere to this definition. In Figure 1, the two domains are: the Abstract Task Domain [BKL93] (the high level user's tasks needed to execute some computation - represented by the eye) and the application domain represented by the ADT. The mapping in Figure 1 is the mapping m in the definition, the input messages are the sentences S and the output is the result of applying the metaphor.

Maiden [Mai91] has discussed the nature of mappings of analogies or metaphors in software engineering. They map only those causal relations belonging to an abstraction shared by the target (the ADT) and the reusable domain (the user's abstract task domain). The causal relations belong to an abstract knowledge structure shared by the reusable and target problems. The ADV concept also conforms to those considerations.

4 ADVs as Module Interconnection Relations

The modular structure of a system can be described in terms of various types of mathematical relations. The entity-relationship model (ER model [Che76]) was motivated by the need for a conceptual model of data suitable for specifying user views and logical requirements in applications that are centered around large collections of interrelated data (databases). The model is based on three primitive concepts: entities, relations and attributes. The properties of an entity are its attributes and the relations in which it participates. Relations may be annotated as one to one, one to many, many to one, or many to many. These annotations describe simple constraints on the relationship.

When we describe general programming systems, as opposed to database-centered applications, we need relations to be constrained in ways that characterize the structure of software designs [Par72b, Par72a, Par76].

Four types of relations among modules are useful for structuring software designs in which decomposition by form (object) with decomposition by function are combined. They are: USES, IS_COMPONENT_OF, COMPRISES and INHERITS_FROM.

For distinct modules M_i and M_j, we say that M_i USES M_j if and only if correct execution of M_j is necessary for M_i to complete the task described in its specification. If M_i USES M_j, we also say that M_i is a client of M_j, since M_i requires the services that M_j provides. In classical structured

design (strict functional decomposition) the USES relation is restricted to a hierarchy. In object oriented design, the USES relations among modules is defined statically, that is, the identification of all pairs $< M_i, M_j >$ belonging to USES (identification of clients and servers) is independent of the execution of the software.

In design, once M_i is decomposed into the set $M_{s,i}$ of its constituents, it is replaced by them, that is, M_i is an abstraction (COMPRISES the set $M_{s,i}$) that is implemented in terms of simpler abstractions (that relate to the abstraction by means of the relation IS_COMPONENT_OF). A module M_i can also be organized as a hierarchy of modules that inherit its properties (INHERITS_FROM), that is, their attributes as objects fall under the same general category.

The relations USES and IS_COMPONENT _OF/COMPRISES provide only a rough description of the software architecture. More remains to be said regarding the exact nature of the interaction between two modules participating in the USES relation and about the details of IS_COMPONENT_OF/COMPRISES.

In our generalization of the ADV/ADT concept we say that an ADV VIEWS an ADT. The VIEWS relation combines the USES and the COMPRISES relations and also provides the additional details regarding the interaction between two modules. The ADV provides the client ADT with the services it needs from a server ADT and it may also express the COMPRISES relation by integrating component ADTs through the nesting of their respective ADVs. INHERITS_FROM also holds within ADVs.

Nesting and inheritance also implement the relations IS_COMPONENT_OF and INHERITS_FROM at the ADT level.

The relation VIEWS was informally described above as a combination of USES and COMPRISES extended to include additional details regarding the interaction between two modules. In what follows we provide examples of the additional details that can be encapsulated in ADVs.

In some object-oriented design methodologies [Boo91, R$^+$91] many different relationships between objects may be realized as client relationships. The relation VIEWS constraints the semantics of the client relationship to mean essentially IS_A_COMPONENT_OF. The server object is completely encapsulated by the client and may therefore be regarded as its private resource.

To exploit the inherent concurrency of object orientation, several categories of objects had their roles identified in designs of concurrent systems. Examples are active "actor" objects [Boo91] that export no methods (their only function is to call other objects in the system), objects which possess no active thread (servers or passive objects) and "agent" objects [Boo91] that both export methods for invocation by others and possess their own thread to initiate calls. Since their methods may be called concurrently by other objects, they are likely to require an appropriate behavior to control external access. The first and third categories are examples of the additional details regarding the interaction between modules that ADVs should be able to capture.

Also, to provide maximum flexibility in the allocation of virtual node instances to physical machines in a network, the distributed run-time system should support the same communication mechanisms for local (intra-processor) and remote (inter-processor) communication - a property known as communication transparency. There is one fundamental aspect of objects that is detrimental to their suitability for acting as virtual nodes - their failure to encapsulate their internal state. Again, this is another example of interaction between modules that may be dealt with by ADVs.

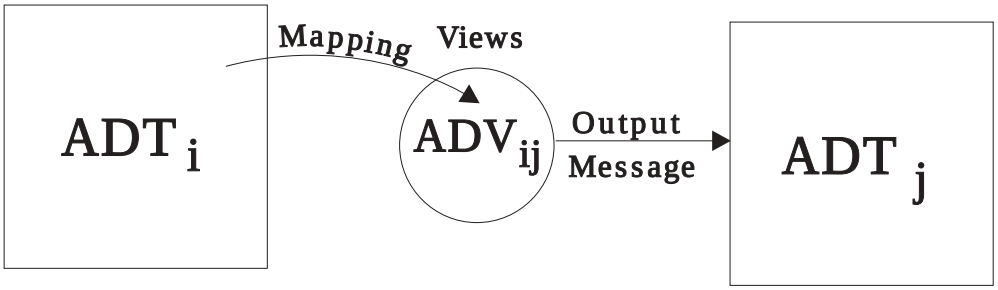

Figure 4: An ADV providing an interface between two ADTs

ADV *name*[**For Type** *adt_map_names*]

Declaration: *var_name*1: *Type*1, *var_name*2: *Type*2, ...

Invariant: *inv-name* $\triangleq$ *formulae*

MESSAGE *Output*[**For Type** *adt_output_names*] ()

⋮

End *name*

Figure 5: An Abstract Data View (ADV) Specification Schema for a module interface

5 ADVs as Domain Transformers

ADVs can be used to transform one semantic domain to another, a process that is usually associated with a stepwise refinement or top-down approach to design. In this section we illustrate this transformation in two steps. First we show how ADVs can be used to hide a portion of the interface of an ADT and thus produce a more specialized version. This concept is illustrated by showing how a deque can be transformed into a queue or stack through a simple application of an ADV. In a second step we show how ADVs can be viewed as metaphors as defined in Section 3.2. Here we discuss the concept and illustrate the approach by transforming the design of a mail system into the lower level functions of a file system.

The first step illustrates the notion of dynamic inheritance supported by ADVs to promote reuse by stimulating the development of general components which can be viewed in various different ways. In other words, with ADVs we are able to mimic in object-oriented design and programming the way generality and reuse could be achieved in data base systems.

The second step relates to our previous work on application integration [CILS93b, CLV93]. Originally we have used external ADVs (user interfaces) to integrate applications. If the application ADT we want to connect to an ADV is expressed at a low level of abstraction, the gap needs to bridged by other layers of ADVs and ADTs.

The argument we have used to consider external ADVs as metaphors also apply to the situation in which a client ADT is viewing a server ADT. At the linguistic level we could name an ADV's messages with the terminology from the problem domain. Thus, ADVs can be made to represent metaphors in the problem domain known to the end-user.

At the design level the ADV concept can be modified to specify completely the relationship between modules represented by ADTs. For example, ADV_{ij} depicted in Figure 4 shows that the client ADT labelled ADT_i views the component ADT labelled ADT_j. In other words the ADV implements the module relations VIEWS (USES and COMPRISES). The ADV can also support the detailed definition of the interface between modules.

A modification of the ADV specification schema from Figure 3 is shown in Figure 5 and this new schema definition supports the concept of an interface between modules. The ADV approach assumes that the client ADT never includes operations related to interfaces of other ADTs which it uses. In our approach, the ADV represents all aspects of the interface of the component ADT which are relevant to the client ADT. In the syntax of the modified schema, the representation relation requires that both the client ADT name and the component ADT name be specified in the ADV. The client ADT name should appear after the key words **For Type** in the ADV declaration, and the component ADT name after **For Type** in the output message.

Whenever the declaration **For Type** is used in both the ADV and output message declarations, two pseudo-variables *client_owner* and *component_owner* become available inside the ADV. The variables *client_owner* and *component_owner* are associated with the client and component ADTs respectively. These pseudo-variables refer to the two ADTs and allow controlled access.

The pseudo-variable *component_owner* connects an instance of the ADV to an instance of the component ADT and provides access to its interface. This method of specifying the connection to the component ADT ensures that the client has no knowledge of the interface of the component ADT. Further it allows the ADV to implement interface hiding or interface specialization where a client ADT may only have access to a subset of the operations provided by a component ADT. Also if the method of interface access provided by the client differs from that supported by the component the ADV can effect a transformation.

The pseudo-variable *client_owner* associated with an instance of the client ADT is made available inside the ADV. This pseudo-variable refers to the client ADT and allows controlled access to the state of the ADT. The client ADT can only be modified through its operations but read-only access to its state can be provided through the pseudo-variable *client_owner*. The *client_owner* pseudo-variable can then be used in the state invariant for the ADV and, in a restricted form, in the post-conditions, thus, allowing the "appearance" of an ADV to conform to the state of an ADT. This connection between the ADT and ADV ensures that the ADV has current knowledge of the state of the ADT.

The schema which supports interfaces between ADTs only contains output message definitions, since there is no active external object to trigger an event which would cause an input message. ADTs do not trigger events and ADVs only connect ADTs together and do not interact directly.

From an operational viewpoint the ADV can be viewed as a process which responds to changes in the client ADT state[1] and responds to these changes by activating the interface of the component ADT. The ADV acts in a manner not unlike a channel in a computer system.

In summary we observe that there are two types of ADVs: an ADV which acts as an interface between two different media and has both input and output messages, and an ADV which acts as

[1] The client ADT changes its state because of a direct or indirect stimulus caused by an input event such as that coming from a user interface, a timer or some other interrupt-driven device.

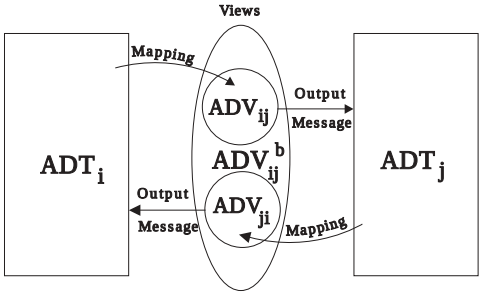

Figure 6: A pair of ADVs providing a bi-directional interface between two ADTs

an interface between two ADTs operating in the same medium.

The interface specified in Figure 4 and 5 only supports uni-directional communication and does not allow the component ADT to return values to the client. In order to allow bi-directional communication we must use a pair of ADVs as depicted in Figure 6. The second ADV named ADV_{ij} provides access to the state of ADT_j and returns values to ADT_i through the output message. These two ADVs are composed into a single ADV called ADV^b_{ij} and this specification schema is outlined in Figure 7.

5.1 A Simple Example of the Use of ADVs as Domain Transformers

In the simple example presented in this section we illustrate the use of ADVs as general interfaces by demonstrating how a double-ended queue (deque) can be presented as either a stack or queue. We first present the specification of the three ADTs, the deque, stack and queue followed by the two ADVs which interconnect the stack to the deque and the queue to the deque. Since the two ADVs are bi-directional this example shows simple nesting of ADVs and many other aspects of their basic structure and yet manages to avoid detail which might obscure the concept.

We start the example by describing the application, that is, by providing a simplified specification for the three ADTs (deque, stack and queue) that encapsulate the application; this description is shown in Figures 8 and 9. Note that we formulate an application specification which is entirely independent of the interface of the components.

In Figure 8 *Element* stands for an abstract element to be placed in the deque. No operations in the example require pre-conditions because the restrictions that the deque is empty is defined in the post-conditions for the *RemoveRight* and *RemoveLeft* functions.

Given the ADT specification for the application, we can now define the two ADVs *Stack* and *Queue* in Figure 10 which connects the ADTs for *Stack* and *Queue* to the ADT for *Deque*. This interface specification performs the primary function of connecting the ADTs and hiding aspects of the *Deque* interface that are not necessary for the specific definition.

The ADTs *Stack* and *Queue* are associated with the ADT *Deque* through the **For Type** operator specified in the ADVs *Stack* and *Queue* and in the output messages in Figure 10. The item shown after the **For Type** operator in the ADV statement is associated with a client and the item after the **For Type** operator in the output message is connected with a component. The pseudo-variables *client_owner* and *component_owner* are made available through this association

ADV ADV_{ij}^{b}

Declaration: var_name1: $Type1, var\_name2$: $Type2, \ldots$

Invariant: $inv\text{-}name \triangleq formulae$

ADV ADV_{ij}[**For Type** ADT_i

$\vdots$

MESSAGE $Output$[**For Type** ADT_j] ()

End ADV_{ij}

ADV ADV_{ji}[**For Type** ADT_j

$\vdots$

MESSAGE $Output$[**For Type** ADT_i] ()

End ADV_{ji}

$\vdots$

End ADV_{ij}^{b}

Figure 7: An Abstract Data View (ADV) Specification Schema for a bi-directional module interface

and allow read-only access to the state of the client and restricted access to the interface of the component. The restrictions imposed on interface access are a function of the ADV.

5.2 ADVs as Metaphors

In the previous section we demonstrated how ADVs can be used as interfaces between two ADTs. This concept can be further generalized, and an ADV can be viewed as a metaphor or transformer between two semantic domains. In terms of design the ADV or metaphor can be viewed as a mapping between two different levels of abstraction in a design. For example, the concept of deleting a file by dragging it to a trash can, can be viewed as a metaphor for deleting a file from a file system, which can be viewed as a metaphor for operations such as locating a file in a directory and altering the various tables. We shall illustrate this concept further by constructing the ADVs or metaphors for an electronic mail example which is illustrated in Figure 11 and has been used as a generic example in the literature [BKL93].

5.2.1 An Example – Electronic Mail

An electronic mail system was chosen to show how a designer might use ADVs to specify the relationship between two different levels within the architecture. Electronic mail can be viewed as operations on a number of ADTs or objects at a level of abstraction viewed by the user. Hence the user would have the concept of mailbox for incoming messages, a letterbox for outgoing messages,

Type *Deque*

Declaration: $componentDeque$: $Element^*$, $DeqEl$: $Element$

Type *Element*

$\vdots$

End *Element*

Function *AddRight* ($DeqEl$)
external wr $DeqEl$
post-condition: $componentDeque = \overleftarrow{componentDeque} \frown [DeqEl]$

Function *AddLeft* ($DeqEl$)
external wr $DeqEl$
post-condition: $componentDeque = [DeqEl] \frown \overleftarrow{componentDeque}$

Function *RemoveRight* () res: $\{empty, DeqEl\}$
external rd $DeqEl$
post-condition: if $ISEMPTY()$
then $res = empty$
else $\overleftarrow{componentDeque} = componentDeque \frown [DeqEl] \wedge res = DeqEl$

Function *RemoveLeft* () res: $\{empty, DeqEl\}$
external rd $DeqEl$
post-condition: if $ISEMPTY()$
then $res = empty$
else $\overleftarrow{componentDeque} = [DeqEl] \frown componentDeque \wedge res = DeqEl$

Private Function $ISEMPTY$ () res: $\mathbb{B}$
post-condition: $res \Leftrightarrow (length(componentDeque) = 0)$

End *Deque*

Figure 8: The ADT for a Deque

Type *Stack*

Declaration: $componentStack$: $Element^*$, $StackEl$: $Element$

Type *Element*

⋮

End *Element*

Function *Push* ($StackEl$)
external rd $StackEl$
post-condition: $componentStack = \overleftarrow{componentStack} \frown [StackEl]$

Function *Pop* () res: $StackEl$
external wr $StackEl$
post-condition: $\overleftarrow{componentStack} = componentStack \frown [StackEl] \wedge res = StackEl$

End *Stack*

Type *Queue*

Declaration: $componentQueue$: $Element^*$, $QueueEl$: $Element$

Type *Element*

⋮

End *Element*

Function *Enqueue* ($QueueEl$)
external rd $QueueEl$
post-condition: $componentQueue = \overleftarrow{componentQueue} \frown [QueueEl]$

Function *Dequeue* () res: $QueueEl$
external wr $QueueEl$
post-condition: $\overleftarrow{componentQueue} = [QueueEl] \frown componentQueue \wedge res = QueueEl$

End *Queue*

Figure 9: The ADTs for a Stack and Queue

ADV *Stack*

ADV *Push*[**For Type** *Stack*]

MESSAGE *Output*[**For Type** *Deque*] ()
external rd *client_owner.StackEl*, wr *component_owner.DeqEl*
pre-condition: $client\_owner.StackEl = component\_owner.DeqEl \wedge$
$client\_owner.StackEl <> empty \wedge$
$client\_owner.Push = component\_owner.AddRight$

End *Push*

ADV *Pop*[**For Type** *Deque*]

MESSAGE *Output*[**For Type** *Stack*] ()
external wr *component_owner.StackEl*, rd *client_owner.DeqEl*
pre-condition: $component\_owner.StackEl = client\_owner.DeqEl \wedge$
$component\_owner.Pop = client\_owner.RemoveRight$
post-condition: $component\_owner.StackEl = client\_owner.RemoveRight$

End *Pop*

End *Stack*

ADV *Queue*

ADV *Enqueue*[**For Type** *Queue*]

MESSAGE *Output*[**For Type** *Deque*] ()
external rd *client_owner.QueueEl*, wr *component_owner.DeqEl*
pre-condition: $client\_owner.QueueEl = component\_owner.DeqEl \wedge$
$client\_owner.QueueEl <> empty \wedge$
$client\_owner.Enqueue = component\_owner.AddLeft$

End *Enqueue*

ADV *Dequeue*[**For Type** *Deque*]

MESSAGE *Output*[**For Type** *Queue*] ()
external wr *component_owner.QueueEl*, rd *client_owner.DeqEl*
pre-condition: $component\_owner.QueueEl = client\_owner.DeqEl \wedge$
$component\_owner.Dequeue = client\_owner.RemoveRight$
post-condition: $component\_owner.QueueEl = client\_owner.RemoveRight$

End *Dequeue*

End *Queue*

Figure 10: The ADVs to connect a Stack and a Queue to a Deque

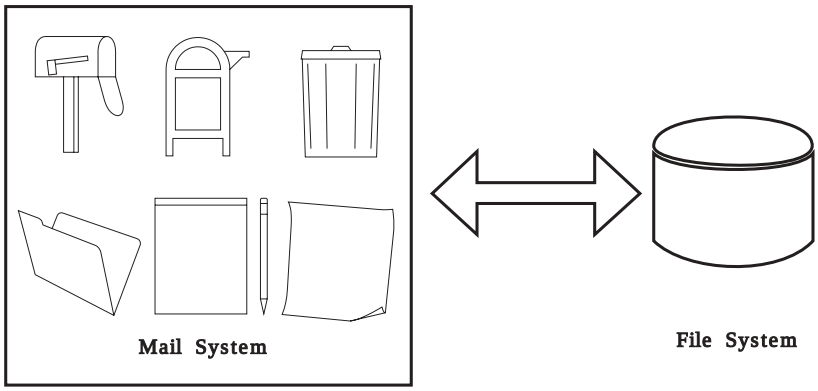

Figure 11: The Mail Example

a trash can for deleted messages, a set of folders on a desktop to retain important messages, a notepad to create a message, and of course, the object message. All of these concepts would relate to operations on the file system. Hence our mail system consists of a number of objects at one level of abstraction or semantic domain which must somehow be mapped into the semantic domain of the file system. This mapping which is a metaphor, is defined by an ADV.

We give a partial illustration of the ADV as a design metaphor by showing how it maps the operations on the object mailbox onto the operations on the file system. Figures 12 and 13 show the two ADTs for the mail system and file system, while Figures 14, 15 and 16 contain the complete definition of the ADV that acts as the metaphor mapping the mail system into the file system.

The function of this ADV is to map the functions of the mail system such as *Select* and *View* into equivalent functions in the file system such as *Access* and *Read*. This is illustrated in Figure 15 in the ADV for Mailbox where the result of the *Mailbox.Select* operation in the ADV *MailSystem* is first transformed from a mailbox name into a file name by the function *string2file*. This name is then used by the file system function *Directory.Access* to return a handle to a directory which is stored in a variable *dirptr* in the enclosing ADV *Mail_System_Metaphor*. The variable *dirptr* is converted from a file handle into a directory name by the function *file2string* and then used as an argument in the *Mailbox View* command. Here we see how the ADV acts as a metaphor; it transforms the commands and their arguments from the semantic domain of the mail system to the semantic domain of files and directories.

6 Implementing ADVs

ADVs were originally conceived as a method of reusing designs for user interfaces and ADTs in highly interactive systems. In the previous section we have generalized the ADV concept to encompass general interfaces between ADTs in any part of a program design. As general interfaces ADVs can be implemented using several different models [CCCL93] such as MVC [KP88] and ALV [Hil92] and at several different times during the program creation process. This section outlines some of the possible implementation strategies.

Type *Mail_System*

Declaration: *componentMailbox*: *Mailbox*

Type *Mailbox*

Declaration: *msg*: *Message*

Type *Message*

Declaration: *hdr*: *Header*
bdy: *Body*

Type *Header*

⋮

End *Header*

Type *Body*

⋮

End *Body*

Function *View_Header* ()
external rd *hdr*
post-condition: "Header displayed"

Function *View_Body* ()
external rd *bdy*
post-condition: "Message displayed"

End *Message*

Function *Select* ()
external rd *msg*
post-condition: "Message selected" $\wedge$ *Header.View_Header*

End *Mailbox*

Function *Select* ()
external rd *componentMailbox*
post-condition: "Mailbox selected"

Function *View* ()
external rd *componentMailbox*
post-condition: "List of messages viewed using Header.View_Header command iteratively"

End *Mail_System*

Figure 12: The ADT for the mail system

Type *FileSystem*

Declaration: *dir*: *Directory*
dir_handle: *Directory_pointer*

Type *Directory*

Declaration: *file*: *File*

Type *File*

⋮

End *File*

Function *Access* ()
external rd *file*
post-condition: "File named file available"

End *Directory*

Function *Access* (*dir_handle*)
external rd *dir*
post-condition: "Directory named dir available"

End *FileSystem*

Figure 13: The ADT for the file system

ADV *Mail_System_Metaphor*

ADV *MailSystem*[**For Type** *MailSystem*]

⋮

End *MailSystem*

ADV *FileSystem*[**For Type** *FileSystem*]

⋮

End *FileSystem*

Private Function *string2file* (*x*: *pointer_to_string*) *res*: *y*: {*empty*, *z*: *pointer_to_file*}
pre-condition: *x* <> *empty*
post-condition: "x converted to z"

Private Function *file2string* (*y*: *pointer_to_file*) *res*: *x*: *pointer_to_string*
pre-condition: *y* <> *empty*
post-condition: "y converted to x"

End *Mail_System_Metaphor*

Figure 14: The ADV acting as a metaphor between the mailbox and the file system

ADV *MailSystem*[**For Type** *MailSystem*]

ADV *Mailbox*[**For Type** *Mailbox*]

ADV *Message*[**For Type** *Message*]

ADV *Header*[**For Type** *Header*]

MESSAGE *Output*[**For Type** *File*] ()
external rd *client_owner.msg*
post-condition: $component\_owner.File.Access(string2file(client\_owner.msg))$

End *Header*

ADV *Body*[**For Type** *Body*]

MESSAGE *Output*[**For Type** *File*] ()
external rd *client_owner.msg*
post-condition: $component\_owner.File.Access(string2file(client\_owner.msg))$

End *Body*

End *Message*

MESSAGE *Output*[**For Type** *FileDirectory*] ()
external rd *client_owner.componentMailbox*
post-condition: $component\_owner.Directory.Access($
$string2file(client\_Owner.componentMailbox))$

End *Mailbox*

End *MailSystem*

Figure 15: The ADV for the Mail System

ADV *FileSystem*[**For Type** *FileSystem*]

MESSAGE *Output*[**For Type** *Mailbox*] ()

ADV *Directory*[**For Type** *Directory*]

ADV *File*[**For Type** *File*]

MESSAGE *Output*[**For Type** *Header*] ()
external rd *client_owner.dir*
post-condition: *component_owner.Header.View_Header(*
file2string(*client_owner.dir*) ∧ "end of header marker")

MESSAGE *Output*[**For Type** *Body*] ()
external rd *client_owner.dir*
post-condition: *component_owner.Body.View_Body(*
file2string(*client_owner.dir*) ∧ "start of body marker")

End *File*

MESSAGE *Output*[**For Type** *Mailbox*] ()
external rd *client_owner.dir*
post-condition: *component_owner.Mailbox.View*(*file2string*(*client_owner.dir*))

End *Directory*

End *FileSystem*

Figure 16: The ADV for the file system

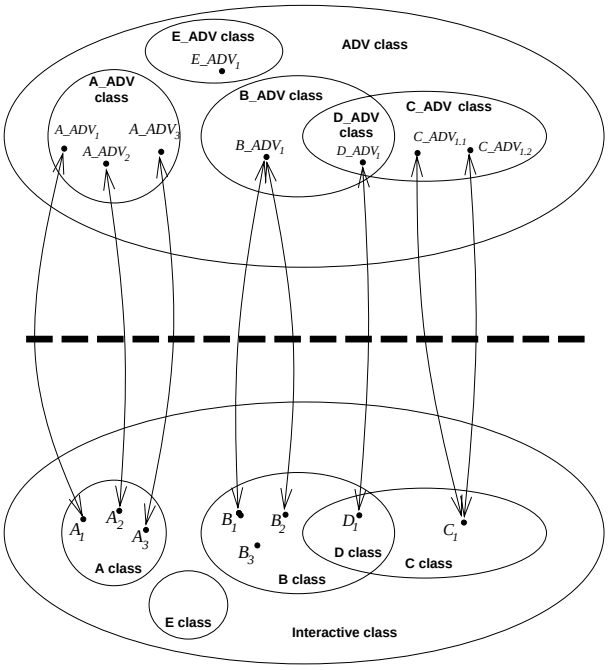

Figure 17: Programming ADV – ADT designs

6.1 ADVs and ADTs as Objects and Classes

One implementation strategy which might be called the single-language[2] approach produces ADVs as modules or objects in the sense of object-oriented programming. As an illustration of this technique we give a brief overview in this Section of how ADVs relate to the ADT classes of object-oriented programming. This approach has already been tested extensively in the implementation of a GUI generator in C++ developed by Potengy and described in [LCP92].

ADT classes which communicate with the user through an associated ADV can be viewed as subclasses of a general Interactive class as illustrated in Figure 17. Each of these ADTs inherits the properties of the general Interactive class, and an instance of this class is an Interactive object. The ADV subclass structure shown in Figure 17 is specified reflecting the Interactive subclass structure, obviously providing access only to its public members.

The ADV class contains at least a Window and a reference to an Interactive object. Each instance of ADV is related only to one Interactive object at a time, but many ADV objects can be related to the same Interactive object at the same time. For example, in Figure 17, the A_ADV instance A_ADV_1 is associated with the instance A_1 of A. The instance B_ADV_1 sometimes is linked to B_1 and sometimes to B_2. There are also two instances of the C_ADV class associated with the same instance C_1 of C. Of course there must be a mechanism to tell the ADV to which instance it is related. However, because the way instances are stored is application dependent, the

[2] We use the term single-language because most of these concepts would be implemented in a single programming language rather than using multiple languages.

application designer will be in charge of specifying the association mechanism.

When an Interactive object is changed by itself, by an ADV, or by any object internal to the application, it will send a message to all associated ADVs, requesting that they update themselves. To do this, the Interactive class contains a list of references to ADV objects. Experience with programming ADVs indicates that they should be able to send messages to each other. This message passing is only necessary with ADVs that are directly or indirectly related to the same object. Thus, the best object to manage the message traffic is the Interactive object. In order to improve the opportunity for reuse, the ADV subclass should be as general as the ADT class with which it is associated, thus, the same abstract operations will be applied to ADVs and their associated ADTs. For example, in Figure 17, class D is a subclass of B and C; since its corresponding ADV is built in the same way, the class D_ADV is a subclass of B_ADV and C_ADV.

Another approach which delays the binding time would be to consider ADVs and ADTs as object modules and connect them together when the modules are linked. Such an approach might require a more complex linker than is normally used, since the linker instructions would likely be ADVs. Except for completeness we do not discuss this approach further.

6.2 ADVs and ADTs as Dynamic Objects and Classes

Another implementation approach could be called the domain-expert approach. In this case a number of application and user interface components are made available. Each component may be quite complex with a well-defined application program interface (API). The API can be linked to and operated dynamically using some form of late binding through a program written in a script language such as the Korn-Shell [Par89], Perl [Wal92], Expect [Lib90], Visual Basic [Cor91], VX·REXX or REXX [Cow90]. In fact the scripts which link the various components together are often ADVs themselves in that they act as metaphors. The domain-expert approach has been used in a limited way in Unix to join together filters and other application units in some form of chain using standard input and output ports. We have used this strategy more extensively in our laboratory to produce complex applications from component ADVs and ADTs. Some examples are presented in [CILS93b] and [CLV93].

In this section we use an example to illustrate some aspects of the domain-expert approach. Consider a relational database containing several numeric data fields which we wish to display in a spreadsheet format. The entire application is controlled by a GUI which has text input fields for the name of the database and the corresponding SQL statements. The system is constructed using one of the user interface toolkits such as VX·REXX or Visual Basic, that is, interface components and various application programs are connected together and communicate through a scripting language.

There are several ADTs and ADVs in the system. The database is an ADT and the spreadsheet is an ADV and ADT combination since it contains both a user interface and the application which stores the data. The GUI objects and their associated scripts to control the application form an ADV. Another ADV which is implemented entirely in the scripting language is the connection between the database and the spreadsheet. This ADV acts as a metaphor since it takes data from the database, transforms the data into a format suitable for the spreadsheet, and then communicates with the spreadsheet. This is one illustration of an entire class of applications which can be constructed using the domain-expert approach. Features of this approach include selective interface

ADV *Concurrent*

Declaration: *lock*: *semaphore*

ADV *Stack*

⋮

End *Stack*

ADV *Queue*

⋮

End *Queue*

End *Concurrent*

Figure 18: The ADVs to connect a Stack and a Queue to a Deque with concurrency mechanisms

hiding and dynamic inheritance. Many examples similar to the one described here have been implemented. In fact, developing methods for producing such applications was one of the motivating factors for creating VX·REXX.

7 Concurrency

The examples in Section 5.1 were used to illustrate how an ADV can be used to specify an interface between two ADTs. This approach to interfaces can be easily extended to handle concurrency, and we use the stack, queue and deque example as an illustration. When the stack or queue ADT accesses the deque they cause a change in state in the deque, hence any such access must be in the form of an atomic transaction.

The specification in Figure 18 and 19 illustrates one method of achieving concurrency using an ADV. Since the Stack and Queue ADVs may not operate independently they are enclosed in another ADV called *Concurrent* which supports concurrency mechanisms. In this example a semaphore variable *lock* is declared which is acted upon by two semaphore functions *up* and *down* equivalent to Dijkstra's P and V operations [Dij65]. The pre- and post-conditions were expanded to ensure that execution would only occur if the deque ADT was not already in use.

Two observations can be made about the method which is illustrated in Figure 18 and 19. Concurrency issues are now carefully isolated, they are not part of the ADT. Rather they describe how the ADT should be used. Also this form of packaging of concurrency concerns bears some relationship to the concept of monitor [Hoa75].

The previous example illustrates the principles of using ADVs in a computation involving concurrency but a more complex example is needed to test the method thoroughly. For this reason we designed and implemented a simple cooperative system: a shared drawing tool that is outlined in this section. In the cooperative system users are allowed to draw simple diagrams and write short paragraphs in their workspace. Each user's actions will be replicated in the other active workspaces. Since many workplaces may be active simultaneously, concurrent user actions may take place and

ADV *Stack*

ADV *Push*[**For Type** *Stack*]

MESSAGE *Output*[**For Type** *Deque*] ()
external rd $client\_owner.StackEl$, wr $component\_owner.DeqEl$
pre-condition: $client\_owner.StackEl = component\_owner.DeqEl \wedge$
$client\_owner.StackEl <> empty \wedge$
$client\_owner.Push = component\_owner.AddRight \wedge down(lock)$
post-condition: $up(lock)$

End *Push*

ADV *Pop*[**For Type** *Deque*]

MESSAGE *Output*[**For Type** *Stack*] ()
external wr $component\_owner.StackEl$, rd $client\_owner.DeqEl$
pre-condition: $component\_owner.StackEl = client\_owner.DeqEl \wedge$
$component\_owner.Pop = client\_owner.RemoveRight \wedge down(lock)$
post-condition: $component\_owner.StackEl = client\_owner.RemoveRight \wedge up(lock)$

End *Pop*

End *Stack*

ADV *Queue*

ADV *Enqueue*[**For Type** *Queue*]

MESSAGE *Output*[**For Type** *Deque*] ()
external rd $client\_owner.QueueEl$, wr $component\_owner.DeqEl$
pre-condition: $client\_owner.QueueEl = component\_owner.DeqEl \wedge$
$client\_owner.QueueEl <> empty \wedge$
$client\_owner.Enqueue = component\_owner.AddLeft \wedge down(lock)$
post-condition: $up(lock)$

End *Enqueue*

ADV *Dequeue*[**For Type** *Deque*]

MESSAGE *Output*[**For Type** *Queue*] ()
external wr $component\_owner.QueueEl$, rd $client\_owner.DeqEl$
pre-condition: $component\_owner.QueueEl = client\_owner.DeqEl \wedge$
$component\_owner.Dequeue = client\_owner.RemoveRight \wedge down(lock)$
post-condition: $component\_owner.QueueEl = client\_owner.RemoveRight \wedge up(lock)$

End *Dequeue*

End *Queue*

Figure 19: The Stack and Queue Components of the Concurrent ADV

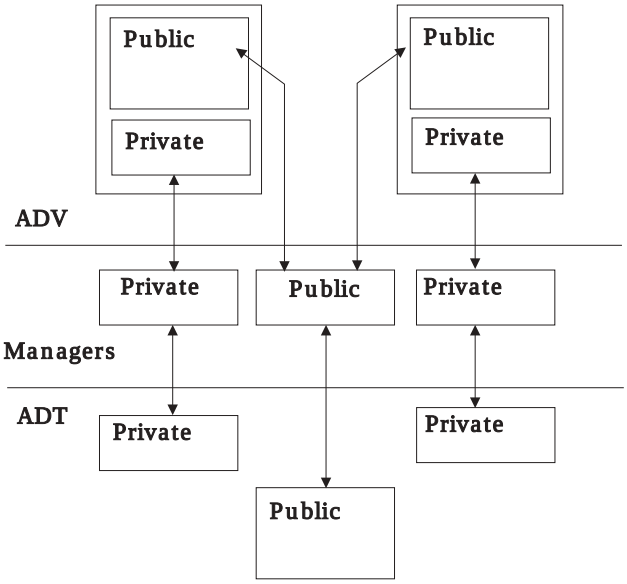

Figure 20: Cooperative Drawing Tool

they should be properly handled.

The design of the concurrent application was simply handled through the use of ADVs. The prototype consists of two windows where squares can be created. Each window represents one screen of a workstation. When a user selects one object in one of the windows, the related object in the other window can not be accessed.

Each workspace has a related ADV which reflects the state of a unique ADT, that maintains the information about all objects drawn by the users. There will also be a private workspace, represented by an ADV related to its own ADT, accessible only to the owner of that workspace. A diagram representing this description is shown in Figure 20.

It is important to note that parallel operations can occur as users work in instances of their workspace. Therefore, since each instance is responsible for its operations, we must be able to guarantee that interference will not occur.

To maintain the consistency of the user's actions we must ensure that each action is applied to all active workspaces. We will use objects called managers to achieve this goal. Whenever the state of an ADT changes, the manager will notify all related ADVs. This configuration is illustrated in Figure 20.

The first time the tool is activated, a public ADT and associated manager will be created in the machine where the program was invoked; this machine is the server. Each time a user joins the session, a new ADV will be created in the user's workstation, which will be "connected" to the manager in the server machine. Only the ADV's visual representation will be exported to the workstations, and all processing will be executed in the server machine.

The managers will maintain a list of dependencies between ADV's and ADT's and any message between them will be handled by the managers. Moreover, whenever there is a relation between an ADT and an ADV there must be a manager.

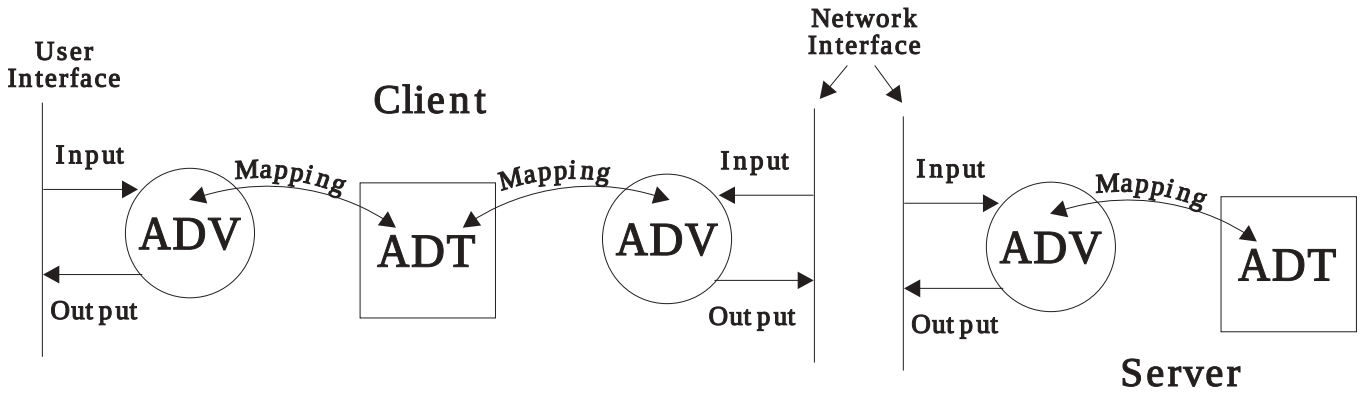

Figure 21: ADVs used to implement the client/server model

The introduction of the manager will result in an intermediate layer between the ADV and ADT world, thus introducing extra work in the implementation phase. This work will be reduced gradually, because the former interfaces and application portions of each implementation can be easily reused.

In order to test the proposed specification, a program was developed using C++ simulating the concurrent operation. Each entity ("object") has at least three related objects in C++: one for the interface part (ADV), another for the application part (ADT) and an object that manages the relations between them.

8 ADVs in Distributed Systems

ADVs are used as an interface between the computer system and the user and have been viewed as a design paradigm for user interfaces. A user interface is one example of a transition between two different worlds, namely the world of the human user and the world inside the computing system. The world of the human user is active, while the computing system world is passive waiting to respond to external commands. The ADV provides a bridge between those two different worlds or media.

There are other examples in computing systems where a bridge for such a transition is required. Networks are one such example. Signals from the network are active commands which require a response from the computer system. This line of reasoning has led to the use of ADVs in other situations such as the one shown in Figure 21. Here two ADVs are acting as interfaces to the network and are supporting the client/server model. The input messages are signals such as datagrams from the network and the output messages respond to changes in the state of the ADT and transmit signals or datagrams to the network. Conceptually there is no difference between the use of an ADV as a user interface or a network interface, and so in a formal specification we use the schema from Figure 3. It is interesting to note that the two ADVs in Figure 21 correspond to the client handler and server handler which are used in client/server systems.

The rest of this section demonstrates how the ADV and ADT concepts map into the client/server environment. We follow this explanation with a discussion of a practical application of this approach

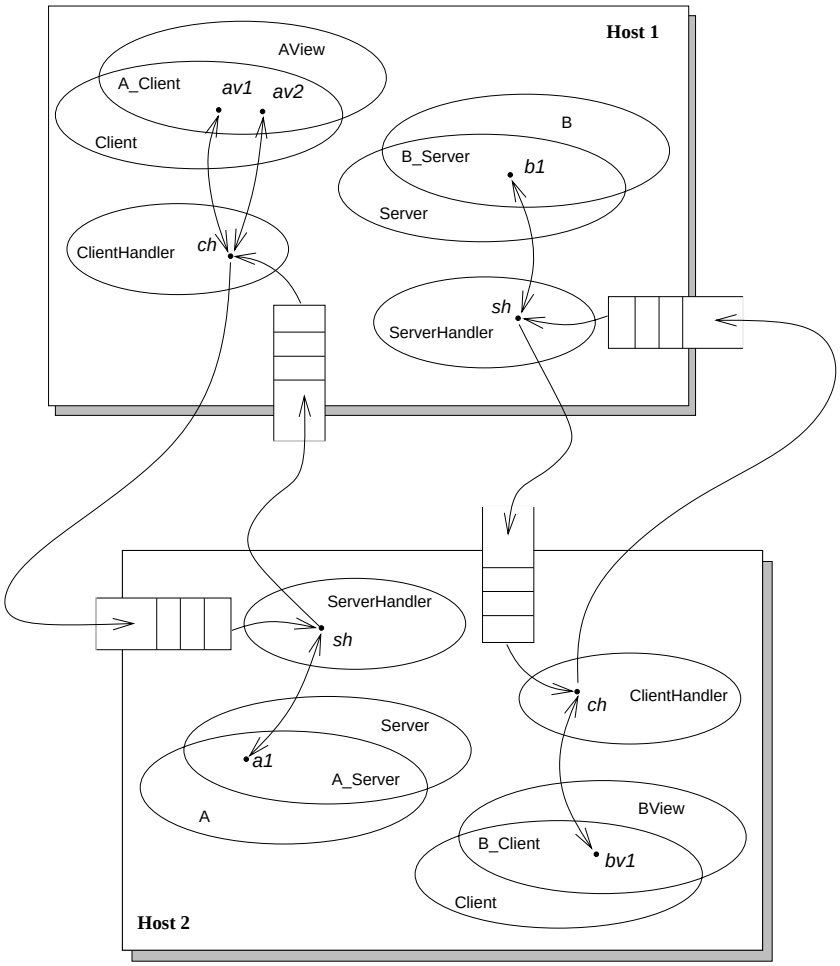

Figure 22: Client/Server architecture using ADVs

by demonstrating a distributed ray tracer.

8.1 The Client/Server Design Environment

The Client/Server environment consists of a set of classes of ADTs and ADVs which are used to build interactive client/server based applications. These classes include the user interface (an ADV) and its associated interactive class [LCP92]that are described in Section 6, a new ADT for the *Server*, and two ADVs for the *ClientHandler* and *ServerHandler*. The *Client* and *Server* are abstract classes, which are specialized through inheritance to perform specific functions.

The *ClientHandler* combined with the *ServerHandler* are responsible for establishing the communication between a *Client* subclass instance and a *Server* subclass instance. A *ClientHandler* is located at the same address space as its associated *Clients* and a *ServerHandler* is located in the same address space as its associated *Servers*.

Of course both the *Client* and *Server* operate without knowledge of their enclosing space. A

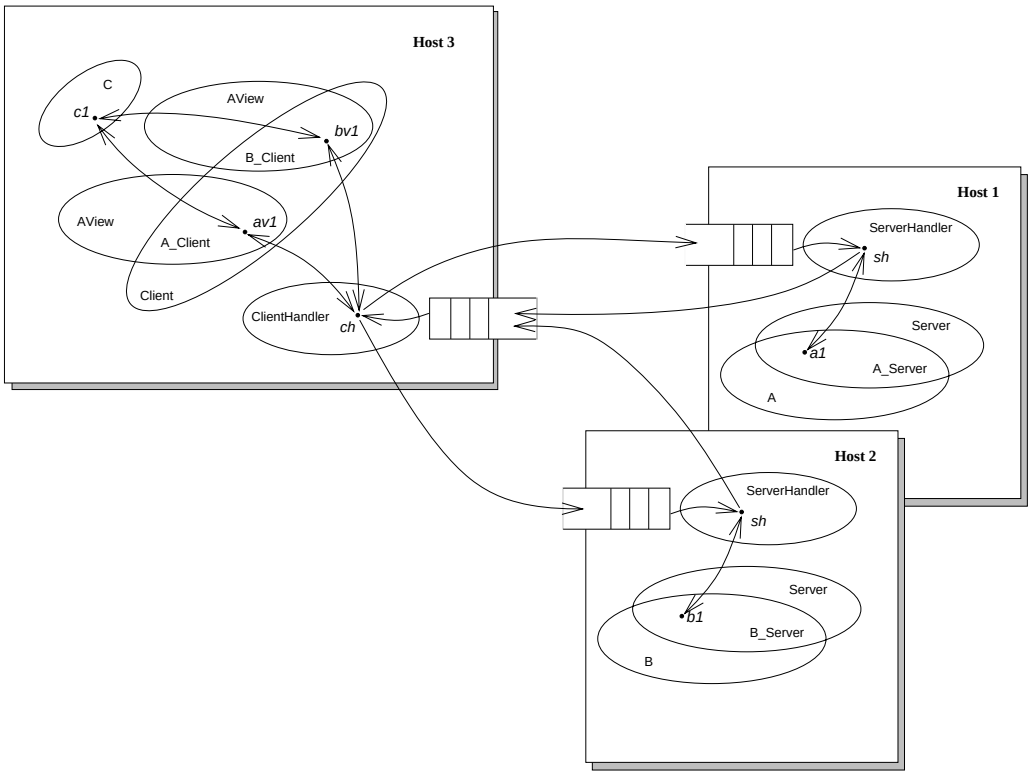

Figure 23: Distributed Object

Client communicates with a *Server* just as if there were no intermediate layer between them. Hence, the handlers must contain maps from their remote customers to the associated remote handlers. For example, if an instance of a class *A_Client* sends a request for an instance of a class *A_Server*, the request is intercepted by the local *ClientHandler*, which delivers the request to the correct *ServerHandler*, based on the *Server* $\times$ *ServerHandler* map in the *ClientHandler*. The *ServerHandler*, in turn, captures the request, forwarding it to the appropriate *Server*. If the *Server* needs to send some reply message, the *ServerHandler* intercepts the reply, delivering it to the correct *ClientHandler*, based on the *Client* $\times$ *ClientHandler* map in the *ServerHandler*. The *ClientHandler* will then forward the reply to the original *Client*. This process is illustrated in Figure 22.

In the Client/Server model, the programmer can specify the remote ADTs without worrying about the lower layers of network or bus communications, or even without considering client/server relations. These issues are postponed until the *Client* and *Server* subclasses are specified.

Returning to the example in Figure 22, we see that the programmer can specify A as an ordinary local object, and *AView* as a simple ADV for A, by extracting its public members. To make it distributed, the programmer need only to create a subclass of A and *Server* (Multiple Inheritance) called *A_Server*, which would act as the server, and a subclass of *AView* and *Client* called *A_Client*, which would act as the client.

This approach supports both modularity and transparency in the context of building distributed

applications. Reuse is also further supported because the same composition operations can be used on the *Client* and the *Server* domains. This model enables its users to create distributed objects as well. As illustrated in Figure 23, the object C is composed of two remote objects A and B, placed in different remote hosts. For the user of object C, everything happens as if C were all local. Simply stated each process has a single client-handler ADV and a single server-handler ADV that jointly handle interprocess communication.

The Client/Server model as implemented in the previous presentation integrates multiple hardware and software systems into a single seamless computing environment. This current research work generalizes the ADV concept beyond user interfaces and makes it into a model for interfaces between ADTs in both a single machine and distributed computing environment.

8.2 An Application: A Simple Distributed Ray Tracer

In order to determine that the ADV concepts were practical in a distributed computing environment we designed and implemented a simple distributed ray tracing algorithm. Ray tracing is a well known concept which is very computing intensive and is an ideal candidate for some form of parallel computing. The ray tracing program consisted of several classes which are mostly illustrated in Figure 24. As well as the objects shown the system required a ray trace manager and a ray tracer. The design and implementation in C++ of the ray tracer which is shown diagrammatically in Figure 25 has been completely successful. The code runs in a client/server environment using both RS6000 and Sun workstations and the performance is more than satisfactory. A description of the details of the design and implementation are provided in [CCLP93].

9 Related Work

The work of D.L. Parnas is one of the major sources of inspiration for the research described in this paper. Parnas pioneered most of the early work on software design: Parnas [Par72b] introduced the concept of information hiding and the notion of module specification [Par72a]. Subsequent work gave additional insights into the issue of program families [Par76] and program modification for extension and contraction [Par79]. In a sense our present work attempts to quantify some of Parnas' earlier ideas. Another early precursor of our ideas of using ADVs to connect objects together is the notion of programming-in-the-small versus programming-in-the-large by De Remer and Kron [DK76].

A number of authors have been working recently on ideas that relate to our proposals. These ideas may be first grouped into the following categories: the use of relations as explicit entities in object oriented design, the use of special classes and objects as interfaces to other classes and objects, and the combination of decomposition by function with decomposition by form.

Rumbaugh et al [R$^+$91] have done important pioneering research in object-oriented design, and in some sense have proposed an up-to-date version of the earlier work by Jackson [Jac83]. They [R$^+$91] propose that it is useful to model a system from three related but different viewpoints: the object model (defines the structure of objects in a system), the dynamic model (describes aspects related to time and sequencing of operations) and the functional model (functions, mappings, constraints and functional dependencies). The methodology that combines the above three views of modelling systems is called the Object Modelling Technique (OMT). The object model of

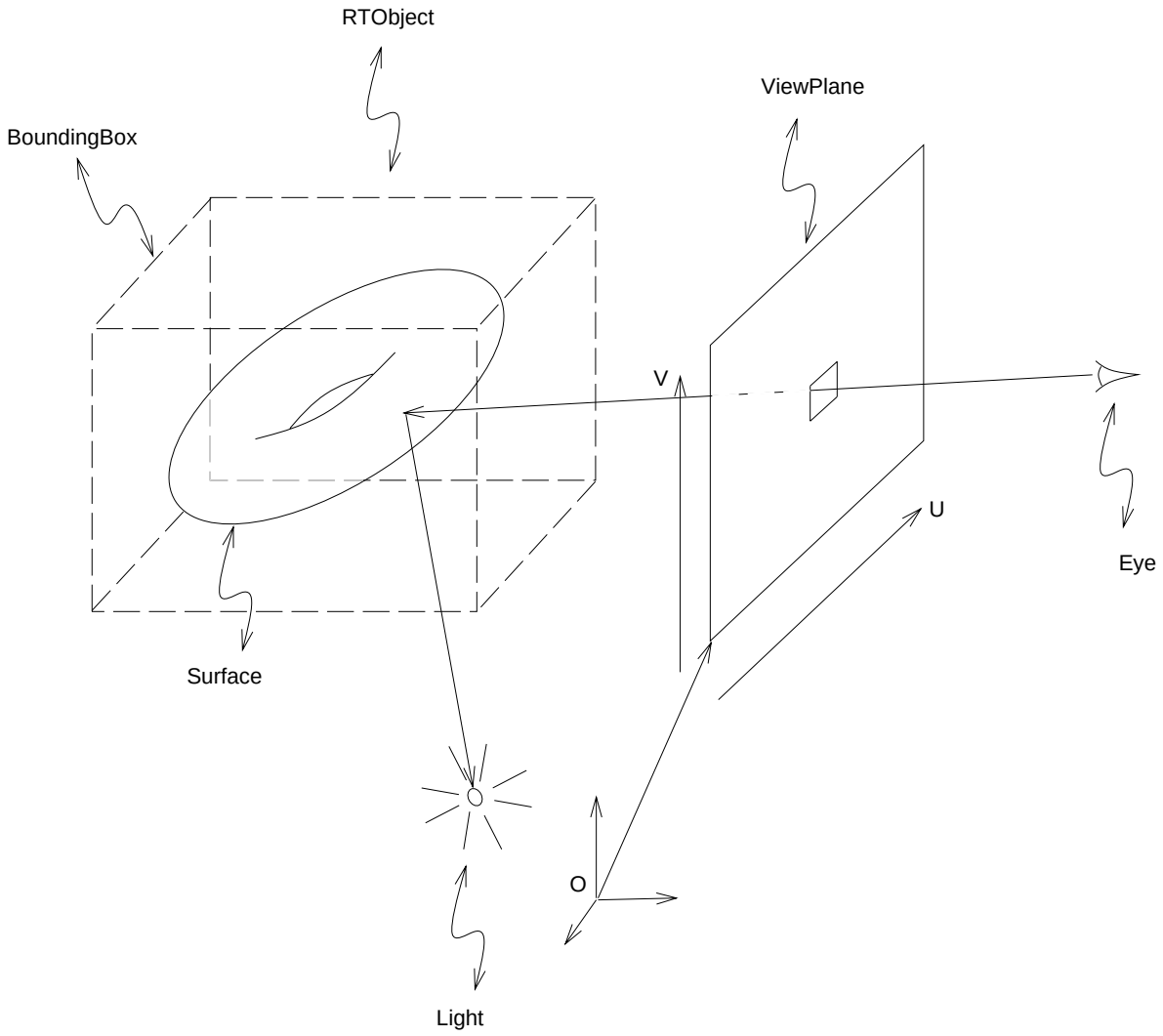

Figure 24: Ray Tracer Objects

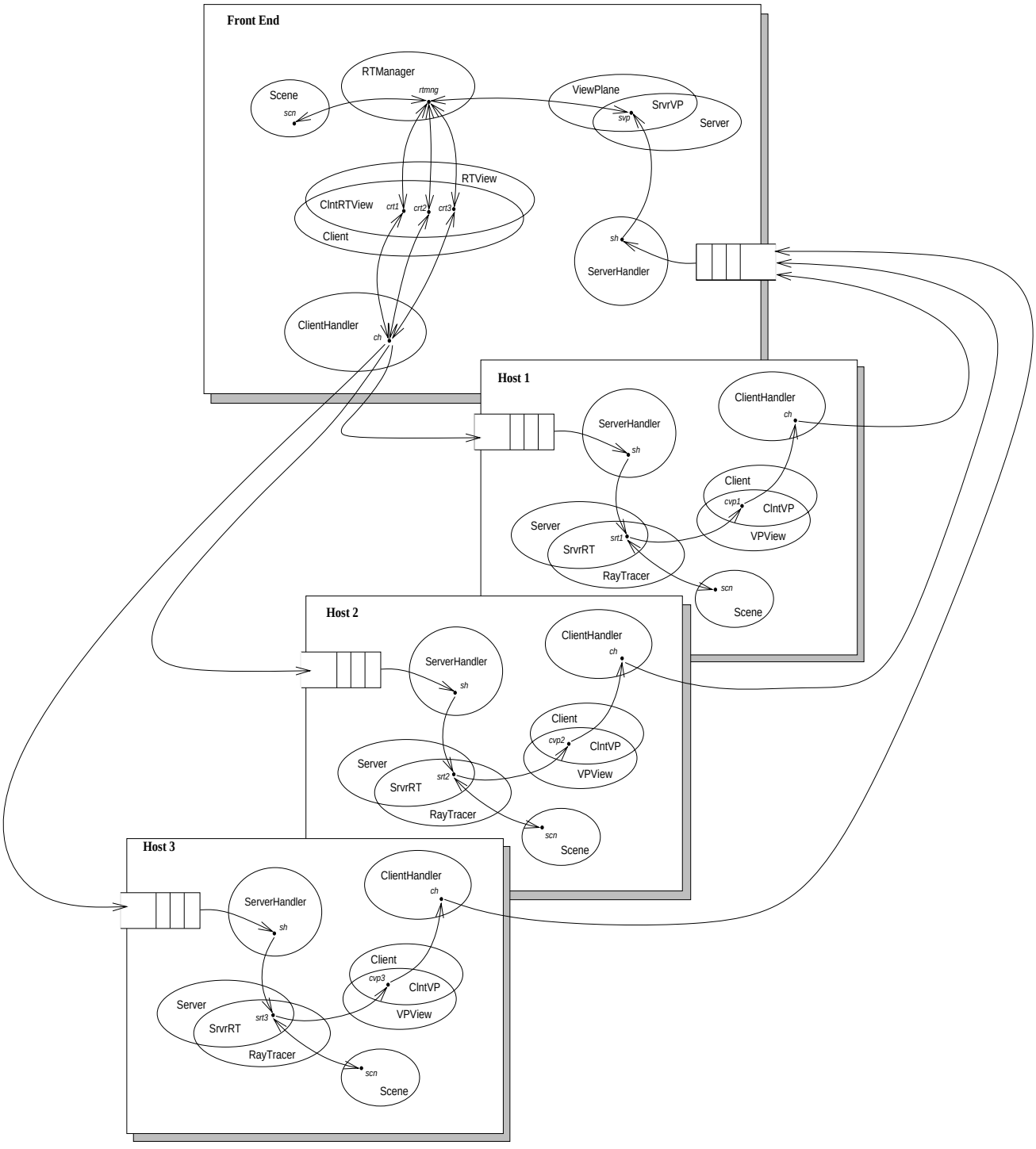

Figure 25: The Distributed Ray Tracer

OMT combines the object-oriented model with the entity-relationship model from data base theory [R+91].

Booch [Boo91] also emphasizes, as does OMT, the need for multiple views of complex software systems. However, he concentrates on decomposition by form (objects) and does not deal with functional decomposition. Booch uses four diagrams called: the class diagram (what classes exist and how are those classes related?), the object diagram (what mechanisms are used to regulate how objects collaborate?), the module diagrams (where should each class and object be declared?) and the process diagram (to what processor should a process be scheduled?). When he discusses relationships among objects, he says that the objects involved in using relationships may play one of three roles: Actor, Server and Agent. Unfortunately he does not appear to elaborate on these roles in the presentation of his methodology. Our ADV proposal may be viewed as an extended combination of the roles of Booch's Actor and Agent objects. Similarly the design primitives provided by ADVs and ADTs could be used to reformulate Booch's object and module diagrams.

One important aspect of our proposal, the balance between decomposition by form and by function, has also been addressed in [Jal89]

Early work by Goguen [Gog86] investigated the reuse and interconnection of software components. He defines a view of an entity A as a theory T, as a mapping from the types of T to the types of A and a mapping from the operations of T to the operations of A, such that the translation of every axiom in T is satisfied by A. The motivation and the context in which his work was developed are similar to the ones that suggested our work but we have arrived at quite different design mechanisms to accomplish similar tasks.

Contemporary work by Kazman et al [BKL93] has proposed that metaphors be considered as first-class abstract data types. They have shown how the specification of metaphors as ADTs constitute the central idea in the conceptual architecture they propose for human-computer interfaces. This previous work uses an approach that parallels our use of ADVs to model user interfaces (an ADV is a special type of ADT) but has been more focused on the issue of formal specification of Human Computer Interfaces as opposed to our more general interest in ADVs as general software design mechanisms. Independent work reported in [SV92] has also suggested that twin ADTs or, more generally, ADTs with multiple representations are a useful concept in the construction of reusable software components for data types.

10 Conclusions

In this paper we have argued the generality of ADVs as a general design mechanism in which both end-users and ADTs VIEW ADTs through ADVs. The proposal of a new design concept requires: formalization to allow for a rigorous understanding of the semantics of the concept being proposed as well as guidelines for future implementation and extensive experimentation (development of several meaningful examples) to validate the underlying ideas. So far we have performed a large number of experiments and have done work on formalization [CCL93]. ADVs at the user-interface level have also been formalized in [CILS93a]. There we have used an extension of VDM [Ier91] that handled the concept of nesting. VDM and ADVcharts (a visual formalism based on statecharts [Har87]) [CCL93] were chosen as formal specification notations because they both provide insights for the development of a methodology based on ADVs and associated tools. We have explored an

approach which should allow us to add the concept of time to the notion of ADV. The introduction of time will allow sequencing to be added to this design concept.

In Section 6 we have discussed two main approaches to the implementation of ADV-based designs: the single-language or object-oriented programming approach and the domain-expert approach. The first approach has allowed us to explore new forms of parallel program design and implementation. In [CCLP93] we discuss the merits of this application of ADVs. The domain-expert approach assumes that a number of application and user interface components are made available as reusable components. We believe this approach using tools such as VX·REXX will enable a new form of program construction we have called Computer Assisted Application Integration [CIS92, CLV93]. Thinking about ADVs as metaphors will play an essential role in this research and it seems clear that results from the general area of domain analysis [Ara93] will also contribute in this context. We believe that the metaphor approach to programming using ADVs may contribute to the problem of reverse engineering of software systems.

A methodology for computer-assisted design based on ADVs will depend on further development of the visual formalism to express the relationships among ADVs, the functional decomposition aspects of ADVs and ADTs as well as timing considerations associated with module interconnections. Extensive experimentation will be required to compare the resulting methodology, for instance, with the ones proposed by Booch [Boo91] and Rumbaugh [R+91]. An ADV-based methodology will provide a rigorous integrated notion of module interfaces that has been missing in previous module interconnection languages.

Many standards in computer/communications such as those proposed by CCITT and ISO emphasize the standardization of interfaces including sequencing. With the addition of time the ADV concept could be used to provide a formal rigorous definition of many of the interfaces in standards documents.

11 Acknowledgements

The authors wish to thank J. Atlee, C. Bicharra, P.J. Bumbulis, L.M.F. Carneiro, S.E.R. Carvalho, M.H. Coffin, R.N. Kazman, and A.B. Potengy for their helpful comments and suggestions on an earlier version of this paper.

References

[Ale64] C. Alexander. *Notes on the Synthesis of Form.* Harvard University Press, 1964.

[Ara93] G. Arango. Domain Analysis Methods. In W. Schoeffer, editor, *Software Reusability.* Harwood, London, March 1993.

[B+92] R. Budde et al. *Prototyping, An Approach to Evolutionary System Development.* Springer-Verlag, 1992.

[BKL93] L. Bass, R. Kazman, and R. Little. Toward a Software Engineering Model of Human-Computer Interaction. In *Proceedings of the Engineering for Human-Computer Interaction*, Amsterdam, North Holland, 1993.

[Boo91] Grady Booch. *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[C+92] D. D. Cowan et al. Program Design Using Abstract Data Views–An Illustrative Example. Technical Report 92–54, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, December 1992.

[CCCL93] L. M. F. Carneiro, M. H. Coffin, D. D. Cowan, and C. J. P. Lucena. User Interface High-Order Architectural Models. Technical Report 93–14, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, 1993.

[CCL93] L. M. F. Carneiro, D. D. Cowan, and C. J. P. Lucena. ADVcharts: a Visual Formalism for Interactive Systems. In *To appear in Proceedings of the York Workshop on Formal Methods for Interactive Systems Springer*, 1993.

[CCLP93] M. Coffin, D. D. Cowan, C. J. P. Lucena, and A. B. Potengy. Distributed Abstract Data Views: Design and Implementation. Technical Report 93-61, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, 1993.

[Che76] P. P-S. Chen. The entity-relationship model: Toward a unified view of data. *ACM TODS*, 1(1):9–36, March 1976.

[CILS93a] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Abstract Data Views. *Structured Programming*, 14(1):1–13, January 1993.

[CILS93b] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Application Integration: Constructing Composite Applications from Interactive Components. *Software Practice and Experience*, 23(3):255–276, March 1993.

[CIS92] D. D. Cowan, R. Ierusalimschy, and T. M. Stepien. Programming Environments for End-Users. In *Proceedings of IFIP 92, Volume III*, pages 54–60, 1992.

[CL93] D. D. Cowan and C. J. P. Lucena. Enhancing Software Design Reuse: Nesting in Object-Oriented Design. Technical Report 93–26, Computer Science Department and Computer Systems Group, University of Waterloo, 1993.

[CLV93] D. D. Cowan, C. J. P. Lucena, and R. G. Veitch. Towards CAAI: Computer Assisted Application Integration. Technical Report 93–17, Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, January 1993.

[Cor91] Microsoft Corporation. *Microsoft Visual Basic Programmier's Guide.* Microsoft Corporation, 1991.

[Cow90] M. F. Cowlishaw. *The REXX Language: A Practical Approach to Programming.* Prentice-Hall, 2nd edition, 1990.

[Das91] S Dasgupta. *Design Theory and Computer Science.* Cambridge University Press, 1991.

[DeM78] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Press, 1978.

[Dij65] E. W. Dijkstra. Co-operating Sequential Processes. In *Programming Languages.* Academic Press, 1965.

[DK76] F. DeRemer and H. Kron. Programming-in-the-large Versus Programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2), 1976.

[Gog86] J. A. Goguen. Reusing and Interconnecting Software Components. *IEEE Computer*, 19(2), 1986.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hil92] Ralph D. Hill. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In *CHI '92*, pages 335–342. ACM, May 1992.

[Hoa75] C. A. R. Hoare. Monitors, An Operating System Structuring Concept. *Communications of the ACM*, 17:549–557, October 1975.

[Ier91] Roberto Ierusalimschy. A Method for Object-Oriented Specifications with VDM. Technical report, Monografias em Ciência da Computação, PUC-Rio, February 1991.

[Ind87] B. Indurkhya. Approximate Semantic Transference: A Computational Theory of Metaphors and Analogies. *Cognitive Science*, 11, 1987.

[Jac83] M. A. Jackson. *System Development.* Computer Science. Prentice-Hall, 1983.

[Jal89] P. Jalote. Functional refinement and nested objects for object-oriented design. *IEEE Trans. on Software Engineering*, 15, 1989.

[KP88] Glenn E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *JOOP*, pages 26–49, August September 1988.

[LCP92] C. J. P. Lucena, D. D. Cowan, and A. B. Potengy. A Programming Model for User Interface Compositions. In *Anais do V Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens, SIBGRAPI'92*, Aguas de Lindóia, SP, Brazil, November 1992.

[Lib90] Donald Libes. expect: Curing Those Uncontrollable Fits of Interactivity. In *Proceedings of the Summer 1990 USENIX Conference, Anaheim, California*, Gaithersburg, MD 20899, June 1990. National Institute of Standards and Technology.

[Mah90] M. L. Maher. Process Models for Design Synthesis. *AI Magazine*, Winter 1990.

[Mai91] N. Maiden. Analogy as a Paradigm for Specification Reuse. *Software Engineering Journal*, January 1991.

[Par72a] D. Parnas. A Technique for Software Module Specification with Examples. *CACM*, 15(5), 1972.

[Par72b] D.L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *CACM*, 15(12), December 1972.

[Par76] D. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2(2), 1976.

[Par79] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138, March 1979.

[Par89] Tim Parker. Shells for UNIX: A Basic Programming Choice. *Computer Language*, 6(7):73, 1989.

[Par90] H. A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[R+91] James Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[SV92] K. Sikkel and J. C. van Vliet. Abstract Data Types as Reusable Software Components; The Case for Twin ADTs. *Software Engineering Journal*, May 1992.

[VRe93] *WATCOM VX·REXX for OS/2 Programmer's Guide and Reference.* Waterloo, Ontario, Canada, 1993.

[Wal92] Larry Wall. *Programming perl.* O'Reilly & Associates, 1992. QA76.73.P347W35x.