Abstract Data Views: Motivation, Current Proposal and Research Trends

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, 22460, RJ,
Brazil
lucena@inf.puc-rio.br

July 28, 1994

Abstract

This paper provides an overview of our current research work on Abstract Data Views (ADVs). ADVs are a new design concept that supports reuse-in-the-large and are a method for specifying and abstracting interfaces between objects. We present the motivation for the problem, express the requirements for an appropriate solution and outline our current proposal. We also provide an overview of the theoretical aspects of ADVs and illustrate how they provide a general solution for the module interconnection problem in object-oriented design.

Resumo

Este artigo apresenta uma vista geral de nosso atual trabalho de pesquisa sobre Visoes Abstratas de Dados ("ADVs"). ADVs são um novo conceito de projeto que suporta reuso em ponto grande e um método para a especificação e abstração de interfaces entre objetos. Nós apresentamos a motivação para o problema, expressamos os requisitos para uma solução apropriada e esboçamos a nossa proposta atual. Nós também apresentamos uma vista geral dos aspectos teóricos de ADVs e ilustramos como eles fornecem uma solução geral para o problema da interconexão de módulos no projeto orientado a objetos.

1 The Problem

A significant barrier to the reuse of both designs and implementations of software objects and modules is the fact that they internalize knowledge about their surrounding environment. For example, a typical module or object of an application often knows about its user interface, specifically details of how its data structures will be displayed, how the user will interact with the application or what objects on the screen correspond to activations of components of the module . Similarly, a module or object knows too much about the services required from other objects or modules. For example, a module will know too much about naming conventions in a file system or about the names of modules from which it acquires services.

Such depth of specialized knowledge seems counter not only to reuse but to good engineering practice in general. There are many ways that a data structure can be displayed, and since this is not an intrinsic property it should not be attached to the data structure. Input has a similar property. There are many ways that a user can interact with an application and so the application should not be aware of the mode of interaction. Similarly a module should know it requires services and specify that fact, but it should not specify how those services are supplied.

What is required is an ability to define an interface that separates the module or object from the user interactions or from the services supplied by another module or object. The interface should be aware of the requirements of the module or object, but the module or object should not be aware of the interface. This approach to defining an interface implies a clear separation of concerns. Such a problem is often addressed in mechanical systems where a linkage "interface" joins two components one of which supplies a service.

2 Current Solutions

Several authors have studied both aspects of this problem, that is the separation of an application from the user interface or in a more general context, the separation of a module or object from its environment. However, we have noticed the two aspects have not been examined in an integrated manner.

Recently, several approaches have been described which examine the clear separation of the user interface from its application. A number of different viewpoints have been used and these are

summarized in the next few paragraphs.

2.1 User Interfaces

A number of authors [CCCL93] have proposed various architectures which state that the user interface should be clearly separated from its application. They have proposed structures that assign various portions of the user interaction and the application to specific components. However, they do not give any guidance on how to design a program to have a reasonable level of assurance that the architecture will be followed.

Other authors have proposed specific implementation strategies that separate the user interface from the application. Examples are the MVC [KP88] and ALV [Hil92] models. For example, MVC uses Smalltalk which has a rich user interface library and ALV uses constraint programming in a LISP environment. Although these are excellent implementation models, it is usually difficult to map these strategies into other programming environments.

Another approach that is often cited, is the use of window toolkits such as X Windows [MA88] and Motif [Ope90]. These systems expose window components as objects which can be accessed by the application. Two issues arise:

1. The code to access a window object typically resides in the application. Hence the code becomes part of the application and the user interface is quite closely bound to the application.

2. The control of user input events is usually in an event dispatcher that is attached to the application code. Thus, adding a new type of input event requires a modification to the application.

In other words many window toolkits do not assist the programmer in clearly separating the user interface and the application. Although it is possible to use these toolkits and maintain a separation, no design guidelines exist on how to achieve this separation.

The problem appears to be that most window toolkits do not support the correct level of abstraction for user interfaces. The toolkits tend to expose details such as the event dispatcher which confuse where the control should be situated. At least one view [CILS93a, CILS93b] is that the control should reside with the user interface and not the application and, that the toolkit should support that view. At least two systems, namely Visual Basic [Cor91] and VX·REXX [VRe93], conform quite closely to this view of user interfaces.

2.2 General Module Interfaces

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.

2.3 Requirements for an Interface Model

What is required is a model for both interfaces and user interfaces that:

1. shows at the design level how to separate both interfaces and the user interface from the application,
2. guides the user into maintaining that separation during implementation, and
3. is independent of specific programming environments and yet can be mapped into any reasonable programming environment.

We have created a design model called the Abstract Data View (ADV) [CILS93a, CILS93b] that we believe has these properties.

3 The ADV Approach

The Abstract Data View (ADV) is a user interface and interface design model that

1. seen as the model of a user interface, represents both the input events and display concepts, that is, all the interactions and communication with the user. Thus, the application has no connection with the user interaction modes except through the user interface.
2. uses the concepts of composition (with locality) of ADVs to represent large aggregates of Abstract Data Types (ADTs) [Hoa89] interconnected through their interfaces to allow for reuse-in-the-large. In composition with locality there are two types of components, external components and internal components. An external component may contain several internal components and an external component may become an internal component. External components may know about the interfaces of their internal components, but internal components do not have any knowledge about their corresponding external components.
3. can be used to transform actions from one media to another media and is a generalization of the role of a user interface. For instance, ADVs can act as a user interface or a network interface. In the case of a network interface one implementation used the ADV as both a client-handler and a server handler.
4. can be used to map the functionality required by an ADT into the functionality provided by another ADT. This is the so-called metaphor.
5. can be used to synchronize access to an ADT from several other ADTs.
6. can be used to hide some of the functionality of an ADT (developed with generality in mind for the purpose of future reuse) providing views to other ADTs (similar to data base design).
7. can be used to interface ADTs that are different representations of the same abstract notion ["twin ADTs"].
8. can be used as "wrappers" that can be placed around an ADT to make it an active process (when in a design ADTs can be seen as passive processes and ADVs as active processes).

9. can be mapped into many different programming environments.

4 ADV Research

DeRemer and Kron [DK76] have coined the terms programming-in-the-small (the coding of individual modules) and programming-in-the-large (programming at the module level), a term that can be used as a synonym for program design. Therefore, by reuse-in-the-large we mean the reuse of program designs involving the description of module specifications and their interconnections. The design concept to be described in this present paper is based on the use of Abstract Data Types (ADTs) [Hoa72] to model program modules.

Using pairs of ADTs to represent modules and their interfaces in reusable designs provided the original motivation for the concept of Abstract Data Views (ADVs) [CILS93a, CILS93b] (pairs of ADTs have also been used in a slightly different sense in [SV92]). A special ADT, called the ADV, is used as an interface (in a very broad sense) for ADTs in designs and provides a view of an ADT. Specification constructors are used to combine ADVs and ADTs to produce more complex designs, and this process has been validated by proof of concept architectures. The approach can be seen as a way of providing language support for the specification and abstraction of inter-object behavior [CL93, HHG90].

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 [PLC93]. 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. ADVcharts a graphical formalism for representing designs using ADVs, have been tested in the production of several different software designs. ADVcharts have also been used to redesign and reengineer an existing interactive software system [Smi94]. In addition, we have shown in [CILS93b] and [CLV93] how ADVs can be used to compose complex applications from simpler ones in a style which is similar to some approaches to megaprogramming [WWC92].

Since we are interested in reuse-in-the-large, we need to combine the specification of modules by guaranteeing syntactic non-interference and semantic context independence [Den73] among the different module specifications. Combination of specifications is achieved by specification constructors [ACCCL94] such as guarded composition (by visibility) multiple guarded composition (sets of components), multiple ordered guarded composition (sequences of components) and inheritance of specifications [Ier91] that are used for both ADV and ADT specifications. ADTs do not know about their associated ADVs; different ADVs may view the same ADT providing the design satisfies a non-interference proof obligation.

We have recently been investigating the incorporation of the notion of an ADV and its specification constructors into VDM [Jon90]. Although VDM seems adequate to model ADV-based reusable designs, we have been using a complementary visual formalism called ADVcharts [CCCL94] based on Statecharts [Har87] and Objectcharts [CHB92], to provide a formal specification or problem statement for the design. Work is in progress on relating these two linguistic representations to allow proof of the correctness of a textual design based on a visual specification. ADVs are also being used to support the formulation of a object-oriented design method; a preliminary version

has been compared to similar efforts reported in the literature such as [Boo91, R$^+$91].

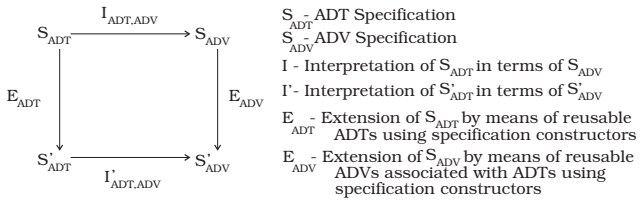

Figure 1: The Modularization Theorem for Reuse of ADTs interpreted through ADVs

We have phrased our central property for reuse-in-the-large in a similar style to the one adopted in [C.86]: *an ADT specification extended by other reusable ADT specifications through the use of a given set of specification constructors can be interpreted (or viewed) as an equivalent ADV specification provided that the original ADT can be interpreted (or viewed) by an associated ADV that is extended by the application of the same given set of specification constructors applied to the ADVs associated with the reused ADTs.* In other words, we need to show that the operators I and E in Figure 1 commute. Using the terminology adopted in [TM87], we can say that we have a "modularization theorem" for the reuse-in-the-large of ADTs interpreted as ADVs.

In a recently published survey [CDG94] a number of formalisms are discussed in the context of reuse. The formalisms address the compositional and the transformational approaches to reuse. The first supports the development of reusable components [Bre91, Ame91, GM91b, GM91a, HN92, Wir88, WHS89] and the second supports the reuse of parts of a development process [C$^+$91a, DS91, C$^+$91b, Rou90]. Our work clearly falls into the first category.

Since the notion of pairs of ADTs (one viewing the other) is central to our work, the logical formalism called "interpretation between theories" [End72] appears as the most adequate choice for its formalization. To make our first version of the theory of ADVs more readable and intuitive, we have resorted in [ACCCL94] to the simplified style of presenting semantics used by Hoare [Hoa72] in his classical seminal paper. For that purpose, we have established the analogy of an ADV as an "abstract program" and its corresponding ADT as a "data representation". In addition, we use states, pre- and post-conditions and invariants in the VDM style [Jon90]. In [Hoa72], knowledge about the semantics of Algol was assumed in order to provide the semantics through "textual substitution". In [ACCCL94], we assume knowledge about the semantics of ADTs, which were originally formalized in [Zil74] and [GTW78].

5 An Informal Overview of the Abstract Data View Concept

The Abstract Data View (ADV) supports a design approach that divides the specification of software modules into two distinct types of components:*Abstract Data Types (ADTs), and Abstract Data Views (ADVs).* An ADV is also an ADT and provides an interface between two ADTs or between an ADT and another medium such as a user or a network. This interface component is called an Abstract Data View (ADV) since it provides a restricted view of an ADT. ADVs were originally conceived as a method of separating the specification of the user interface (UI) components from the non-user-interface (NUI) components of an interactive system and were intended to

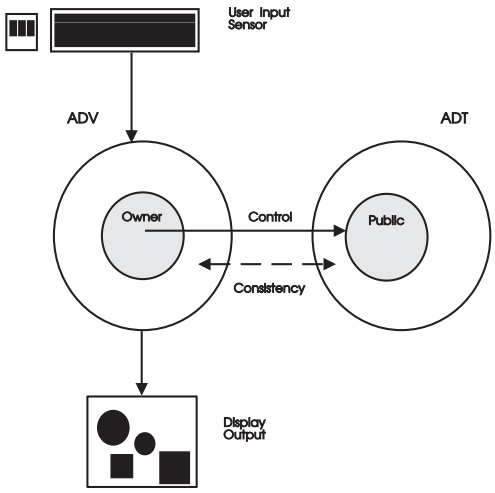

Figure 2: The ADV Model

promote design reuse. Section 6 illustrates the use of ADVs through a number of examples. In the ADV concept, ADTs are completely independent of the interface and have their requirements for data translated by one or more ADVs. In the context of user interfaces this implies that ADTs do not have direct access to input or output.

A typical design of an interactive system that is based on the ADV concept, consists of a collection of ADTs that manage the data structures and state of the NUI part of the system, a collection of ADVs that comprise the perceivable behavior, the UI part of the system, and mappings from ADVs to ADTs.

An ADV description for a user interface contains definitions of all input- and output-handling functions, definitions of all visual objects, and functions for manipulating these visual objects. In an ADV, we describe all design decisions related to the information exchanged between the user and the UI application, among other ADVs, and between the NUI and the UI applications. Figure 2 shows the relationship between input and output devices, ADVs, and ADTs. The relationship between ADV and ADTs, shown as an arrow, associates the public interface of the ADT with the corresponding specification in the ADV. The name of the corresponding ADT in the ADV specification is represented by a variable called *owner*[1] , which provides the connection or "channel" between the UI and NUI parts of the system. ADVs for more general interfaces are specified in a similar way, although there are no input-handling functions specified directly.

The relation between an ADV and an ADT is not symmetric, there is no variable *owner* in the ADT. Since an ADT has no knowledge of input or output, it does not know about its corresponding ADVs. In a design using the ADV concept, a consistency property needs to be satisfied, since several ADV instances could be associated with the same ADT instance in order to provide different views or a different control functionality for the same ADT.

The ADV design concept promotes reuse of interface specifications because it allows complex interface components to be built from simpler interface components. Since ADVs represent ADTs,

[1] because an ADV is associated with an ADT, we say that the ADT is the owner of the ADV

reuse of ADTs is a consequence of this fact. To build complex interface components we use specification constructors based on composition with locality, sets and sequences of components, and inheritance of specifications [Ier91].

6 An Overview of ADVs as a General Design Concept

The Abstract Data View (ADV) concept supports the specification of interfaces between media and between Abstract Data Types (ADTs). In this context, ADVs can be viewed as domain transformers and domain constrainers [CL93] as described in Section 3.

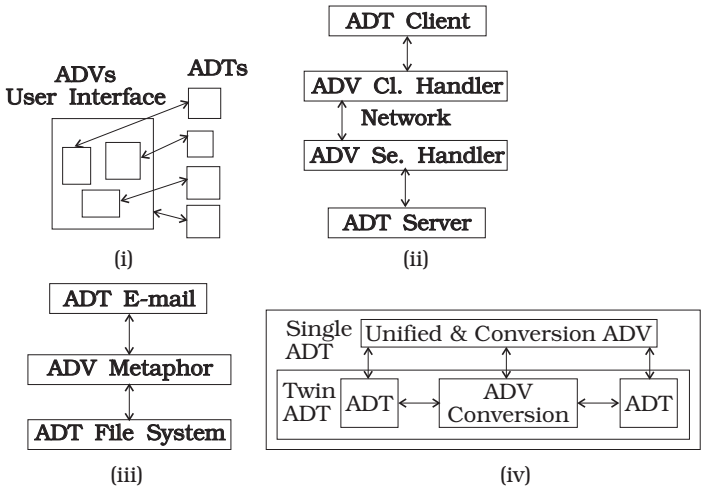

Figure 3: ADVs as Domain Transformers

Some examples of the use of ADVs as domain transformers are illustrated by the four pictures in Figure 3. When an ADV is used to specify a user interface as in Figure 3 (i), ADVs represent all the interactions and communication with the user, and the application has no connection with the user interaction modes except through the user interface. Thus ADVs in this context act as a transformer between the "outside" domain of the user and the "inside" domain of the application or ADTs. Of course, the user interface can also be viewed as a metaphor for the familiar user actions. The concept of an ADV transforming an "outside" domain to an "inside" domain can be generalized beyond user interfaces. This concept is shown in Figure 3 (ii) where ADVs are shown as interfaces to a network in the form of Client and Server Handlers.

The concept of ADV as a metaphor can also be generalized beyond the user interface; ADVs can be used to map the functionality required by an ADT into the functionality provided by another ADT. This concept is also illustrated in Figure 3 (iii) where the ADV is used to map some of the functionality needed by an electronic mail system into the functionality provided by a file system. ADVs can also be used to specify the conversion between two ADTs that are different representations of the same abstract notion ("twin ADTs" or domain-oriented virtual machines [SV92], and also to define a common interface to make the twin ADTs appear as a single ADT. Such a concept is illustrated in Figure 3 (iv).

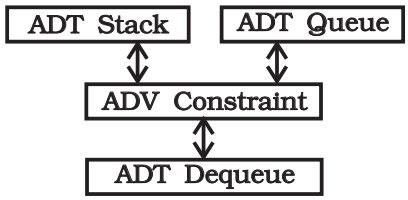

Figure 4: ADVs as Domain Constraints

Although ADVs can be used as general interfaces and metaphors, the ADV can also have added features that can constrain the domain of an ADT. Figure 4 illustrates two ways that this constraint mechanism might be specified. The ADV can be used to hide some of the functionality of an ADT. For example, the diagram shows the ADV making the dequeue ADT appear as both a stack and a queue. Such an approach provides different views of an ADT and is similar to the concept of views in data base design. The ADV can also contain concurrency constraints and limit access to the ADT dequeue by the ADT stack or the ADT queue.

7 Conclusions

There have been a number of software development methods or partial methods proposed over the years. Some chief examples are: Structured Analysis [DeM78, PM84, You89] and Design [WM85, YC79], JSD [Jac83], Object-Oriented Analysis [CY91] and Design [Boo91, WBWW90], and Rumbaugh [R$^+$91]. All these methods have brought some rigour to the process of creating, and maintaining software systems for a broad variety of applications.

We propose to extend the rigour of these and other methods by creating a software development approach in which most steps in the design process can be documented using formal methods. Further, it is intended that this approach should support reuse of designs. Although we have concentrated on the specification of designs, we believe our approach can be extended to provide a formal basis for many of the reification methods that are needed to produce an implementation from a design specification. We have chosen formal methods as the basis for our approach to design specification because it allows us to reason about these specifications, and also build tools that will assist this reasoning process and support transformations between different levels of design specification.

References

[ACCCL94] P.S.C. Alencar, L.M.F. Careiro-Coffin, D. D. Cowan, and C.J.P. Lucena. Towards a Formal Theory of Abstract Data Views. Technical Report 94–18, University of Waterloo, Computer Science Department, Waterloo, Ontario, April 1994.

[Ame91] P. America. *Formal Techniques for Parallel Object Oriented Languages*, volume 612 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

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

[Bre91] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments*, volume 562 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.

[C.86] Gaudel M. C. *Towards Structured Algebraic Specifications*, volume 1 of *ESPRIT 85 Status Report.* North Holland, Brussels, 1986.

[C+91a] J. Cazin et al. *Construction and Reuse of Formal Program Developments*, volume 494 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.

[C+91b] J. Cramer et al. Towards a Formally Based Component Description Language - a Foundation for Reuse. *Structured Programming*, 12(2):91–110, April 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.

[CCCL94] L. M. F. Carneiro-Coffin, D. D. Cowan, and C. J. P. Lucena. ADVcharts: a Visual Formalism for Highly Interactive Systems. In M. D. Harrison and C. Johnson, editors, *Software Engineering in Human-Computer Interaction.* Cambridge University Press, 1994.

[CDG94] J. Cramer, E. E. Doberkat, and M. Goedicke. Formal Methods in Software Reusability. In W. Shafer, R. Prieto-diaz, and M. Matsumoto, editors, *Software Reusability.* Ellis Horwood Limited, 1994.

[CHB92] D. Coleman, F. Hayes, and S. Bear. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering*, 18(1), 1992 1992.

[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.

[CL93] D. D. Cowan and C. J. P. Lucena. Abstract Data Views: A Module Interconnection Concept to Enhance Design for Reusability. Technical Report 93–52, Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, November 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.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Yourdon Press, Prentice-Hall, 1991.

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

[Den73] J. Dennis. Modularity. In *Advanced Course on Software Engineering.* Springer-Verlag, 1973.

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

[DS91] E. E. Doberkat and H. G. Sobottka. A Set-Oriented Program Description Language for ADA. In *Proceedings of the 1st International Workshop on Software Reusability*, pages 193–197, June 1991.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[GM91a] M. C. Gaudel and T. Moineau. Software Reuse Through Formal Specifications. In *Proceedings of the 1st International Workshop on Software Reusability*, June 1991.

[GM91b] M. C. Gaudel and T. Moineau. *A Theory of Software Reusability*, volume 300 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.

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

[GTW78] J. Goguen, J. Thatcher, and E. Wagner. An Initial Algebraic Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 4. Prentice-Hall, 1978.

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

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *OOPSLA'90*, pages 169–180, 1990.

[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.

[HN92] R. Hennicker and F. Nickl. A Behavioral Algebraic Framework for Modular System Design with Reuse. Technical report, Department of Computer Science, Ludwig-Maximilians University, Munich, 1992.

[Hoa72] C. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1(4):271–281, 1972.

[Hoa89] C. A. R. Hoare. Proof of Correctness of Data Representations. In *Essays in Computer Science*, pages 103–115. Prentice-Hall, 1989.

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

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

[Jon90] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, second edition, 1990.

[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 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.

[MA88] Joel McCormack and Paul Asente. An Overview of the X Toolkit. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, pages 46–55, October 1988.

[Ope90] Open Software Foundation. *Application Environment Specification(AES) User Environment Volume*, 1990.

[PLC93] A. B. Potengy, C. J. P. Lucena, and D. D. Cowan. A Programming Approach for Parallel Rendering Applications. Technical Report 93–62, Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, March 1993.

[PM84] J. Palmer and S. McMenamin. *Essential Systems Analysis.* Yourdon Press, Prentice-Hall, 1984.

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

[Rou90] F. Rouaix. Safe Run-time Overloading. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, New York, 1990. ACM.

[Smi94] D. Smith. Abstract Data Views: A Case Study Evaluation. Technical Report 94–19, University of Waterloo, Computer Science Department, Waterloo, Ontario, April 1994.

[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.

[TM87] W. M. Turski and T. S. E. Maibaum. *The Specification of Computer Programs.* Addison-Wesley, 1987.

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

[WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.

[WHS89] M. Wirsing, R. Hennicker, and R. Stahl. *An Example for the Systematic Reuse of Specifications*, volume 387 of *Lecture Notes in Computer Science.* Springer-Verlag, 1989.

[Wir88] M. Wirsing. Algebraic Description of Reusable Software Components. In *Proceeding COMPEURO 88.* Computer Society Press of the IEEE, 1988.

[WM85] P. T. Ward and S. J. Mellor. *Structured Development for Real-Time Systems.* Yourdon Press, Prentice-Hall, 1985.

[WWC92] G. Wiederhold, P. Wegner, and S. Ceri. Towards Megaprogramming. *CACM*, 35(11), November 1992.

[YC79] E. Yourdon and L. Constantine. *Structured Design.* Prentice-Hall, 1979.

[You89] E. Yourdon. *Modern Structured Analysis.* Prentice-Hall, 1989.

[Zil74] S. Zilles. Algebraic Specification of Data Types. Project MAC Progress Report 11, MIT, 1974.