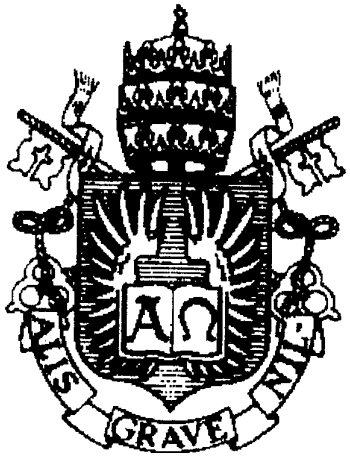
PUC



ISSN 0103-9741

Monografias em Ciência da Computação
nº 35/94

Towards a Logical Theory of ADVs

Paulo Sergio C. Alencar
Luiza M. F. Carneiro-Coffin
D. D. Cowan
Carlos José Pereira de Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, N° 35/94

Editor: Carlos J. P. Lucena Dezembro, 1994

Towards a Logical Theory of ADVs *

Paulo Sergio c. Alencar **

Luiza M. F. Carneiro-Coffin ***

D.D Cowan ***

Carlos José Pereira de Lucena

* Research partly sponsored by the Ministério de Ciencia e Tecnologia da Presidência da República.

** Departamento de Ciência da Computação, Universidade de Brasília, Brasilia, Brazil.

***Computer Science Department & Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada.

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio — Departamento de Informática
Rua Marquês de São Vicente, 225 — Gávea
22453-900 — Rio de Janeiro, RJ
Brasil
Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645
E-mail: rosane@inf.puc-rio.br

Towards a Logical Theory of ADVs

P. S. C. Alencar * L. M. F. Carneiro-Coffin † D.D. Cowan ‡ C.J.P. Lucena §

Abstract

In this paper we motivate and describe a preliminary logical theory for Abstract Data Views (ADVs) and their associated Abstract Data Objects (ADOs). Abstract Data Views (ADVs) define interfaces between application components (ADOs) or between application components and an "external" environment such as a user or a network. ADVs are ADOs that have been augmented with event-driven input and output operations and a mapping that ensures an ADV interface conforms to the state of its associated ADO. Different ADVs may view or interact with the same ADO since interfaces can support alternate "views" of data or modes of interaction, providing the design satisfies certain consistency obligations. In order to maintain separation of concerns and promote design reuse, ADOs are specified to have no knowledge of their associated ADVs. ADVs can be seen as a way of providing language support for the specification and abstraction of inter-object behavior. The logical approach identifies the individual formal units or objects (ADOs and ADVs) that compose the specification of a system with theories in a temporal logic with a (global) discrete linear time structure. In this logical formalism, objects are characterized by their signatures, thus enabling local reasoning to be performed over components of a system. We also adopt an open semantics so that objects are not taken in isolation but can interact with surrounding objects, a pre-requisite for object composition. Mappings or morphisms are used as a means of connecting ADOs and ADVs thus formalizing the views relationship and other ADO and ADV specification constructors. Finally, we present a proof outline of a modularization theorem of ADOs interpreted through ADVs in the context of this logical formalism.

1 Motivation

This paper addresses the question: "How do we provide a formal basis for a practical approach to designing object-oriented software systems?" We have already created a software design model, called the Abstract Data View (ADV), which models the interfaces to highly interactive software systems and emphasizes the separation of concerns between the interface and the application component. We have shown that this ADV model can be used to specify the functionality of interfaces to other system components as well as user interfaces.

Abstract Data Views (ADVs) [13, 14] are Abstract Data Objects (ADOs) that have been augmented with event-driven input and output operations and a mapping that ensures an ADV interface conforms to the state of its associated ADO. Different ADVs may view the same ADO since interfaces can support alternate "views" of data or modes of interaction, providing the design satisfies certain consistency obligations. In order to maintain separation of concerns and promote reuse, ADOs are specified to have no knowledge of their associated ADVs. ADVs can be seen as a way of providing language support for the specification and abstraction of inter-object behavior [15, 29].

ADVs have been used to support user interfaces for games and a graph editor [11], to interconnect modules in a user interface design system (UIDS) [34], to support concurrency in a cooperative drawing tool, and to design and

*P. S. C. Alencar is a Visiting Professor in the Computer Science Department at the University of Waterloo, Waterloo, Ontario, Canada and is currently on leave from the Departamento de Ciência da Computação, Universidade de Brasília, Brasília, Brazil. Email: alencar@csg.uwaterloo.ca

†L. M. F. Carneiro-Coffin is a PhD candidate in the Computer Science Department at University of Waterloo and holds a doctoral fellowship from CAPES. Email: lmfcarne@neumann.uwaterloo.ca

‡D. D. Cowan is a Professor in the Computer Science Department at the University of Waterloo, Waterloo, Ontario, Canada. Email: dcowan@csg.uwaterloo.ca

§C.J.P. Lucena is a Visiting Professor in the Computer Science Department at the University of Waterloo, Waterloo, Ontario, Canada and is currently on leave from the Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Brazil. Email: lucena@csg.uwaterloo.ca

implement both a ray-tracer in a distributed environment [41] and a scientific visualization system for the Riemann problem. A research prototype of the VX-REXX [1] system was motivated by the idea of composing applications in the ADV/ADO style. ADVcharts have been tested by using them to produce several different software designs and to redesign and reengineer the Rita text editor, an existing interactive software system [17, 43, 8]. In addition, we have shown in [14] and [16] how ADVs can be used to compose complex applications from simpler ones in a style which is similar to some approaches to component-oriented software development and megaprogramming [46].

Software has traditionally been designed using one of two basic divide-and-conquer paradigms: decomposition by form or decomposition by function. Decomposition by form implies dividing an object into smaller objects, and decomposition by function implies dividing an action into a number of actions. We believe that both paradigms are needed in the software design process as is also the case in classical engineering design. Decomposition by form is currently supported by object-oriented design methods such as those proposed by Booch [5] or Rumbaugh [42], while decomposition by function is supported by Structured Design as described by Yourdon and others [45, 49]. The approach proposed by Rumbaugh supports both form and function, but they are treated distinctly. We have proposed a unified design method that incorporates both decomposition by form and decomposition by function.

Our design approach constructs software system designs using composition operators and two types of component specifications: application components (ADOs) and interface components (ADVs). The composition operators constitute a formally defined set of specification constructors that operate on ADVs and ADOs to produce composite ADVs or ADOs from simpler components, or sets or sequences of simpler components. The constructors defined by our design method support decomposition by function, while decomposition by form is supported by the fact that both the instantiations of ADOs and ADVs are objects. The semantics of these constructors and ADVs have been chosen to encourage reuse of design specifications through separation of concerns. The composition operations are guarded composition by inclusion and composition through inheritance. Guarded composition by inclusion implies reuse of syntactic components, whereas composition by inheritance implies modification of the component syntax. Composition by inclusion is guarded to enforce constraints on what information a component type is able to access from its composite type; everything a component type must know about the composite type is provided through parameters at instantiation.

Software designs must also express other types of relationships or constraints among the components. For example, the many-to many relationships that arise in entity-relationship designs for databases. Rumbaugh's design approach allows expression of these types of relationships. Relationships among the component ADOs in our approach can be expressed by specializing an ADO with the addition of an invariant or by the use of an invariant in an ADV component, since invariants provide a powerful mechanism for expressing constraints. For example, the fact that an arc in a graph must be terminated by two nodes can be expressed in an invariant.

Both ADOs and ADVs can be specified using textual and graphical formalisms, and a mapping exists between the two forms of expression. The current textual formalism is based on object-oriented versions of VDM[33] and VDM++ [20] (an extension that handles concurrency). The present graphical formalism called ADVcharts [7, 6] is used for designing user interfaces, and is based on state-machine approaches similar to Statecharts[28] and Objectcharts[10]. We have created a graphical formalism because visualizing the design is easier, and sequential and time relationships are more readily apparent. Both the textual and graphical representations support the specification of structure and behaviour. Structure of designs is provided by the composition operators, while behaviour is expressed through either temporal logic or state machine formalisms.

We have chosen formal methods as the basis for our approach to design specification because they support unambiguous specification, allow for tool-supported mathematical reasoning about these specifications and could allow rapid prototyping of functionality. Formal methods and reuse are closely related; reusing a component developed using formal methods spreads the higher development costs over more projects while improving both the quality of the development process and resulting software systems.

Many papers referenced in this paper are available via anonymous ftp from [csg.uwaterloo.ca] at the University of Waterloo. The names of the papers are in the file "pub/ADV/README" and the papers are in the directories "pub/ADV/-demo", "pub/ADV/theory", and "pub/ADV/tools".

2 Formal Methods

We have devised a set of specification constructors to build complex ADOs and ADVs from simpler ADOs and ADVs. Designs using the ADV/ADO approach and these constructors can be expressed in an extended version of VDM or using ADVcharts. We have recently defined a formal semantics for the specification constructors. Using these semantics we can show that applying a constructor to a complex ADO or ADV only augments its previous properties. In this context, we have proven a modularization theorem that shows that a composition of ADOs obtained through a sequence of applications of the constructors to these ADOs is equivalent to the composition of the ADVs related to these ADOs when the constructors are applied in the same order. The proof of the modularization theorem about reuse through ADVs is presented in [3, 2], and was obtained by using a semantics for the constructors given in terms of textual substitution in a manner similar to that used by Hoare [30].

The modularization theorem can be stated more precisely as: an ADO specification extended by other reusable ADO 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 ADO 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 ADOs (see Figure 1). Using the terminology adopted in [44], we can say that we have a "modularization theorem" for the reuse-in-the-large of ADOs interpreted as ADVs. Although in early papers about ADVs we have used the term ADT [30], in order to avoid misunderstandings we introduced the term ADO (because the ADO is an ADT with state memory and actions or methods to change this state memory).

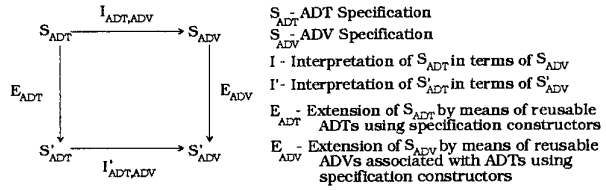


Figure 1: The Modularization Theorem for Reuse of ADOs interpreted through ADVs

The dynamic properties of ADVs can be expressed using a temporal logic formalism. Using this formalism we are able to talk about instances of ADVs and ADOs and analyze consistency between an ADV and an associated ADO (vertical consistency) and between several ADVs associated with the same ADO (horizontal consistency). These results on reuse and consistency have been reported in [3, 2].

Although we have achieved these results, our formalization does not allow us to reason about the properties of designs. Since ADOs and ADVs can be understood in terms of mathematical theories and ADVs view or interpret ADOs, we plan to use "interpretation between theories" [21] expressed in logic in order to produce an integrated theory of ADOs and ADVs with accompanying inference rules.

Similarly the ADVchart graphical formalism has been interpreted in terms of statecharts (one form of operational semantics). An operational semantics approach does not provide the ability to reason about the interactive properties of human interfaces. Since statecharts have been successfully formalized before in terms of denotational semantics [31], we also plan to use denotational semantics accompanied by inference rules to express the formalism represented by ADVcharts.

Using these formal bases for ADVs we will gain the advantages mentioned previously and also should be able to reason about interactive systems and interface designs as suggested in [35, 37].

3 A Preliminary Logical Theory of ADVs

3.1 A General Overview of the Theory

The notation in which the preliminary (non-logical) semantics is expressed in the preliminary version of the theory ([3, 2]) was chosen for its readability thus providing a clear understanding of the intuitive aspects of the concepts. In this next step we go further by capturing the semantics in logic by using interpretation between theories. This kind of formalization which was also used in [44], proved very useful in the formulation of a similar modularization theorem (in their case, related to program reification), and will allow us to reason about reusable designs. We now indicate how this style of logical formalization can be used as a formal basis for the ADV approach.

Our logical approach identifies the individual formal units or objects (ADVs and ADOs) that compose the specification of a system with theories in a logic that can be used for supporting object-oriented specifications. We use this method because although the logical approach advocates that the process of program development consists of the manipulation of specification theories, it is much easier to manipulate presentations of theories in which only the (non-logical) axioms of the theory are presented. Specifically, the approach to be used is a proof-theoretic one based on a combination of temporal logic and category theory [24, 25] that was initially developed for the purpose of formalising modularization techniques for reactive systems.

We will adopt a temporal logic with a (global) discrete linear time structure that is close to those used in [40, 4, 36], thus allowing easier assessment of the support for modular specification that will be described. We will also use the fact that temporal logics may be defined that satisfy, to some extent, institution [27] and, hence, that temporal theories may be used as modularization units for concurrent system specification. We emphasize here that the logics that can be used for supporting object-oriented specification are somewhat more complex than a pure temporal logic, and involves deontic notions as well as action modalities besides temporal operators [22, 23, 26]. However, the structuring mechanisms are independent of the underlying logic and, hence, it is simpler to formalize objects as structuring units for concurrent systems specification based on a simpler logic than using all the logical machinery that is more appropriate for object description. Neverthless, there are many similarities between the full logical apparatus and the one based only on temporal logic and therefore we can think about temporal logic specification units as objects in the object-oriented sense.

In this logical formalism, signatures are used as a way of localizing change by having a logical role in the formalization of encapsulation, thus enabling local reasoning to be performed over components of a system [24, 25]. We also adopt an open semantics so that objects are not taken in isolation but can interact with surrounding objects, a prerequisite for object composition. Mappings or morphisms are used as a means of connecting ADOs and ADVs thus formalizing the views relationship and other ADO and ADV specification constructors. Using these notions we suggest how to construct a calculus of ADV/ADO systems based on these specification constructors expressed in logic and how to use this calculus as a structuring mechanism for verification.

As a first step, following the theory of institutions [27], we introduce the notions of component signature and, for each signature, interpretation structure, language, axioms (formulae) and satisfaction (truth) relations between interpretation structures and formulae. We have to describe the temporal fragment of the language and present the notion of locality (data abstraction, encapsulation of a set of attributes). Furthermore, we have to present an ADV/ADO description (that will be formed by a component signature and a set of formulae that are the axioms of the description). We also have to describe how to reason about local descriptions of or particular object description of ADVs and ADOs,

As a second step, we have to show how we can assemble ADV/ADO descriptions in order to form the description of more complex system components. In other words, we want to give a formal semantics in logic for the ADO and ADV specification constructors for composition, inheritance, sets and sequences. We shall see that this logical formalism characterizes not only the difference between the specification constructors such as the difference between composition and inheritance, but also allows us to characterize the specification constructors in more detail. For example, it will be possible to differentiate between inheritance and particularization.

In the first step we have to define the syntax and semantics of one chosen logic, and show how a (primitive) system component can be described as a theory presentation. Then, we have to show how such descriptions can be put together in order to provide the specification of systems of interconnected components, and how concurrency is supported. In

this context, we have to define a category of object descriptions that is finitely co-complete and we have to show how the interaction between objects can be expressed through diagrams. We have also to show how to use the diagrams to describe the joint behavior of objects including their interaction.

Having developed a collection of theories that act as descriptions of objects, we can use these theories to describe a more complex system by assembling as nodes of a diagram as many instances of these theories as required, and by using morphisms between these theories as edges of the diagram in order to establish the required interconnections. If the category is "well behaved" that is finitely cocomplete, such a diagram may be collapsed to another theory that provides the description of the complex system as an object itself. This new description can then be used in order to derive properties of the global system, or as a node in another diagram (a component of another system). Hence, we have to provide a notion of morphism between theory presentations and a resulting category that allows for such colimits to be computed.

The morphisms between theory presentations define the notion of structure that the formalism provides for specification [24, 25]. As structure-preserving mappings, morphisms establish the relationship that must exist between two object descriptions so that one of them may be considered as a component of the other. Intuitively, this corresponds to the notion of interpretation between theories. When we use the notion of institution, this relationship consists of a translation between the languages of the two descriptions (a signature morphism) such that the theorems of one are translated to theorems of the second one. That is, we take the preservation of theorems (the meaning of a theory presentation) as the criterion for considering that a theory presentation describes a component of a system described through another theory presentation. It is also possible to restrict the properties that we actually want to preserve along a morphism, namely only safety or liveness properties, leading to different notions of structure. Actually, when we adopt the pure temporal logic formalism, we require the preservation of both safety and liveness properties. In this context, results such as the modularization theorem for reuse of ADOs interpreted through ADVs can be established.

Finally, we show how we can obtain the joint behavior of the interconnection of many individual object descriptions of ADVs and ADOs through the adopted specification constructors and how the concurrent aspects of the resulting descriptions can be treated. The joint behavior can be given in terms of the colimit in the category of object descriptions of the diagram involving the individual descriptions. We should notice here that theories act as "types" in the sense that several instances of the same theory may be used in the same diagram to express the fact that there are several components of the system that are of the same "type" and have the same behavior. We shall see how we can distinguish between these various instances. In other words, a theory act as a "generic" object. We also have to show how to derive global properties, that is, properties of the society or interacting collection of objects. In fact, being an object description, the description of the joint behavior of the individual descriptions has its own properties besides the properties of each individual object description. In particular, an ADV object description has its own properties besides the properties of its associated ADO and its component ADVs.

3.2 The Description of ADVs and ADOs as Logical Objects

In this section we describe the adopted logical formalism,i.e. the logic that will be used to formalize the specification of the individual system components in the object-oriented context. We also show how ADVs and ADOs are represented and how they are interpreted in this logical approach as logical theories, and how it is possible to reason about local descriptions of ADVs and ADOs.

Both ADOs and ADVs can be represented as objects and can have their state changed. However, ADVs are modified ADOs that responds for every interface role inside a software system. Although we can find many structural similarities in both concepts, it is important to observe that there is a clear separation between the ADO world and the ADV world.

In what follows we summarize a mathematical characterization of ADVs and ADOs. According to out logical theory, ADVs and ADOs are characterized by a tuple $< DT, AT, AC, AX >$, where DT represents the usual ADT data type (DT) signature (a set S of sorts and a set F of functions provided by the component), AT represents the attributes of the component, AC represents the actions performed on the component, and AX is a set of axioms that describe the properties of the component. The attributes are represented as a set of entities that can change over time and AC is a set of actions that can report the values of attributes (or query) or change these attributes. causal actions and effectual

actions. Causal actions are the input events provided by an ADV. Effectual actions can be provided by both ADOs and ADVs, but an effectual action can only be caused directly or indirectly by a causal action. Thus, in an ADV-ADO configuration only input events (causal actions) can change the state of the system. In other words, a causal action can make an effectual action occur, but an effectual action can not make a causal action occur. This requirement characterizes the ADO-ADV interaction. Since causal actions come from outside the system, causal actions can not cause other causal actions to occur. If we visualize a tree of actions occurring over time, then a causal action can only appear at the root of a tree of action and this syntactic rule also characterizes the ADO-ADV interaction.

We have been using the pair $DT =< S, F >$ in our component specifications for establishing the types and operations which are available on the values which can be stored in the attributes. We do this because we want to support (abstract) specification of the space of values as well, not just the behavior of the components. If we assume that all ADVs/ADOs operate over a fixed collection of data types (integer, arrays, etc.) then we do not need that component in a signature.

Definition (ADO Description).
An ADO description is a pair $< SG_{ADO},\ AX_{ADO} >$ where SG_{ADO} is an ADO signature and AX_{ADO} is a (finite set of SG_{ADO}-formulae, i.e. the formulae in the temporal language that uses the symbols in SG_{ADO} as atoms (the axioms of the description). ■

Definition (ADO Signature)
An ADO signature SG_{ADO} is a tuple $< DT, AT, AC >$ where DT is an usual ADT signature, AT is a $S^* \times S$-indexed family of attribute symbols, and $AC =< AC_Q, AC_E >$ is a S^*-indexed family of action symbols in which the first set is related to query actions and the second to effectual actions. ■

Each ADV description is also defined as a theory presentation in the chosen logic. The ADV description consists of signature (defining the particular vocabulary symbols that are relevant for the description of the object), and a collection of formulae of the the language generated by from the signature (the extralogical axioms of the description).

Definition (ADV Description).
An ADV description is a pair $< SG_{ADV},\ AX_{ADV} >$ where SG_{ADV} is an ADV signature and AX_{ADV} is a (finite set of SG_{ADV}-formulae, i.e. the formulae in the temporal language that uses the symbols in SG_{ADV} as atoms (the axioms of the description). ■

Definition (ADV Signature)
An ADV signature SG_{ADV} is a tuple $< DT, AT, AC >$ where DT is an usual ADT signature, AT is a $S^* \times S$-indexed family of attribute symbols, and $AC =< AC_C, AC_Q, AC_E, AC_O >$ is a S^*-indexed family of action symbols that are related to causal (input), query (report), effectual (that are consequence of other actions) and output actions, respectively. ■

The DT (data type of universe) part contains the state-independent information, i.e. it states the information that does not change with time. It also gives the data context in which the object is placed. From a logical perspective, the universe parameters have a rigid interpretation. The AT (attribute structures) part, on the other hand, contains the information that is state-dependent. This part may include, for instance, program variables, database attributes, or frame slots. A nullary attribute symbol correspond to a program variable, whereas a non-nullary attribute symbol can be associated with data structures. The attribute parameters have a non-rigid logical interpretation. The AC (action structures) part include the data used to define the effects of the actions on the attributes (also known as methods) or to be transmitted (received) as part of a communication (interaction with other objects). Actions have an special role in the formalization of the notion of locality. The elements of the ADV signature θ are assumed to be disjoint and finite.

Let us now describe how the SG signatures are interpreted. We note that a system unit can be an ADV or an ADO and we denote a system unit by SU. A semantic interpretation structure for a system unit signature is defined as follows.

Definition (SG-interpretation Structures).
A SG-interpretation structure for a signature $SG =< DT, AT, AC >$ is a triple $\mathcal{SG} =< \mathcal{DT}, \mathcal{AT}, \mathcal{AC} >$ where:

1. $\mathcal{DT}$ is an usual ADT DT-algebra, i.e. to each sort symbol $s \in S$ a set $s_{\mathcal{DT}}$ is assigned, and to each function symbol $f \in F_{<s_1,\ldots,s_n>,s}$ a function $f_{\mathcal{DT}} : s_{1\mathcal{DT}} \times \ldots \times s_{n\mathcal{DT}} \longrightarrow s_{\mathcal{DT}}$ is assigned;
2. $\mathcal{AT}$ maps $f \in AT_{<s_1,\ldots,s_n>,s}$ to $\mathcal{AT}(f) : s_{1\mathcal{DT}} \times \ldots \times s_{n\mathcal{DT}} \times N_0 \longrightarrow s_{\mathcal{DT}}$. We use $\mathcal{AT}(f)(i)$ for $i \in N_0$ to denote the function $\lambda(b_1,\ldots,b_n).\mathcal{AT}(f)(b_1,\ldots,b_n,i)$;
3. $\mathcal{AC}$ maps $g \in AC_{<s_1,\ldots,s_n>}$ to $\mathcal{AC}(g) : s_{1\mathcal{DT}} \times \ldots \times s_{n\mathcal{DT}} \longrightarrow \mathcal{P}(N_0)$. ■

Note that according to these interpretation structures we are assuming the natural numbers as the underlying temporal domain, i.e. we are assuming a linear and discrete time. The mapping $\mathcal{AT}$ defines the "state" at each time instant, i.e. it gives the values of the attributes at each point in time. The mapping $\mathcal{AC}$ defines for each action symbol the set of instants during which instances of the action occur (it is a predicate over time instants). In other words, the interpretation of the action symbols gives for each instant the set of actions that take place during that instant. The actions in each such a set are assumed to occur concurrently during that instant. However, the ability of actions to occur concurrently can be restricted through the axioms of a specification that define their effects on the attributes or the restrictions and requirements on their occurrence.

In order to discuss the encapsulation and open semantics principles, we need to formalize the semantic structures intended for this logic. We assume a locality requirement (encapsulation): objects have an intrinsic notion of locality according to which only the actions declared in an object can change the values of its attributes.

Definition (Loci).
Given a SG-signature $SG = < DT, AT, AC >$ and a SG-interpretation structure $\mathcal{SG} =< \mathcal{DT}, \mathcal{AT}, \mathcal{AC} >$, let $\mathcal{E} = \{i \in N_0 \mid i \in \mathcal{AC}(g)(a_1,\ldots,a_n)$ for some $g \in AC_{s_1,\ldots,s_n}, < a_1,\ldots,a_n > \in s_{1\mathcal{DT}} \times \ldots \times s_{n\mathcal{DT}}\}$. Note that the set $\mathcal{E}$ is the set of instants during which an action of the object occurs, i.e. these instants denote state transitions in which the object will be engaged. These transitions are said to be witnessed by SG. The SG-interpretation structure $< \mathcal{DT}, \mathcal{AT}, \mathcal{AC} >$ is said to be a ***locus*** if and only if, for every instant $i \notin \mathcal{E}$ and for every $f \in AT, \mathcal{AT}(f)(i) = \mathcal{AT}(f)(i+1)$. ■

In other words, the values of the attributes of the object can only be changed during witnessed transitions, i.e. due to the ocurrence of one of the actions of the object. The formalization of locality is to be understood by the notion of SG-locus and the notions of object description and morphism between descriptions (to be presented later in this section). In fact, we want a formalization of locality as a criterion to have composable object descriptions as specification modules, for which the notion of SG-locus given above provides a correct model-theoretic basis. The notion of morphism of object descriptions will capture the desired notion of encapsulation so that an object cannot refer to the attributes of another object (for "read" or "write" purposes) without incorporating it as a component. This means that it is not allowed for attributes to be observed "outside" an object, only by objects of which it is a component. The fact that we are dealing with an open semantics is reflected in the possibility that there may be $i \notin \mathcal{E}$, meaning that no action of the component occurs during the transition to $i+1$. Furthermore, we note that the intended synchronous, multiprocessor architecture is reflected in the fact that several actions may be performed during the same transition.

Thus, the semantics of encapsulation is captured by a sub-class of interpretation structures which we call ***loci***. It means that if the object remains idle, the attributes remain invariant.

The notion of object descriptions is given using a temporal logic approach ([4, 24, 25, 36, 40]). The adopted temporal logic language is defined with the usual temporal connectives for future: **G** (always), **F** (eventually), **U** (until) and **X** (next). However, we can assume that a formula ϕ is valid iff it is satisfied at every instant. In fact, validity-based consequence gives rise to necessitation rules of the form $\phi \; \mathbf{G}\phi$ as, intuitively, the specification of a component should consist of formulae that are to hold at any possible time during its life. In order to refer to the initial instant we introduce

the atom $\boxed{\text{BEG}}$ which is satisfied only at time 0.

Definition (Classification).
Assume that ξ is a collection of variables (distinct from the attribute and action symbols). Given a SG signature, we define a classification Ξ as a partial function $\xi \longrightarrow S$ (where S is the set of sorts of the universe signature). We denote by Ξ the S-indexed set given by $\Xi_s = \{x \in \xi \mid \Xi(x) = s\}$. ■

Definition (Interpretation of Terms).
Given a SG-interpretation structure $\mathcal{SU} = < \mathcal{DT}, \mathcal{AT}, \mathcal{AC} >$, a classification Ξ, and an assignment Π for Ξ (mapping each set Ξs to $s_{\mathcal{DT}}$) each term $t \in TR_\theta(\Xi)_s$ defines a mapping $[t]^{\Theta,\Pi} : N_0 \longrightarrow s_{\mathcal{DT}}$ (its interpretation). ■

Definition (Validity of a Formula).
A formula ϕ is true in a SG-structure if and only if it is satisfied by every assignment Π and instant i. ■

Definition (Model of a SU Description).
Given a SU description $< SG, AX >$, where SG is a signature and AX is a (finite) set of SG-formulae, we call a SG-locus that makes all the formulae of AX true a model of the SU description. ■

3.3 Local Reasoning About ADVs

In this section we describe how to reason about local descriptions of ADVs and ADOs, i.e. how to reason "inside" an ADV or an ADO. We give some preliminary definitions and present the locality reasoning rules.

Definition (Validity of an Assertion).
If Φ is a (finite) set of formulae over a signature SG, and ϕ is also a formula over SG, $(\Phi \Rightarrow_{SG} \phi)$ is an assertion. An assertion is valid if and only if every SG-locus that makes all the formulae in Φ true also makes ϕ true. ■

If we have a particular description $SU =< SG, AX >$, we can abbreviate (by using the names of the SU descriptions) assertions of the form $(AX_{SU} \Rightarrow_{SG(SU)} \phi)$ by $(SU \Rightarrow \phi)$.

The properties of a SU description $< SG, AX >$ are just the formulae ϕ for which we can prove that an assertion $(AX \Rightarrow_{SG} \phi)$ is valid, i.e. they are theorems of the theory presentation $< SG, AX >$ associated to the SU. Note that all the formulae in AX are assumed to be satisfied at all times so that the formulae ϕ for which $(AX \Rightarrow_{SG} \phi)$ is valid state properties that hold at any time during the evolution of the SU described by $< SG, AX >$.

Definition (Locality requirement).
The locality requirement is captured by the following axiom: for every signature $SG =< DT, AT, AC >$,

$$\Rightarrow_{SG} ((\bigvee_{g \in AC} (\exists x_g) g(x_g)) \vee (\bigwedge_{a \in AT} (\forall x_a)(\mathbf{X}a(x_a) = a(x_a))))$$

where for each symbol u, x_u is a tuple of variables (all distinct) of the appropriate sorts. This formula tells us that either the next transition will be performed by one of the actions of the SU description, or else all the attributes will remain invariant. ■

From this axiom, it is possible to derive a rule that will assist us in proving safety properties under locality. But first, let us define a set of local terms:

Definition (Local Terms).
Given an ADV signature SG and a classification Ξ, we define inductively the S-indexed set of local terms $LT_{SG}(\Xi)$ as follows:

1. If $\Xi(x) = s$, then x is a term in $LT_{SG}(\Xi)_s$;
2. If $f \in F_{<s_1,\ldots,s_n>,s}$ and t_i are terms in $LT_{SG}(\Xi)_{s_i}$, then $f(t_1,\ldots,t_n) \in LT_{SG}(\Xi)_s$;
3. If $f \in AT_{<s_1,\ldots,s_n>,s}$ and t_i are terms in $LT_{SG}(\Xi)_{s_i}$, then $f(t_1,\ldots,t_n) \in LT_{SG}(\Xi)_s$; ■

Note that we omit the temporal operator.

Definition (Locality Rule).
Let ϕ be a formula and $t_1,\ldots,t_n$ local terms over Ξ. For each $g \in AC$, let x_g be a tuple of variables of the appropriate sorts, and assume that they do not occur in Ξ. Then

$$((g(x_g) \rightarrow \phi \mid g \in AC),\ (\bigwedge_{1 \leq i \leq n} (\mathbf{X}t_i = t_i) \rightarrow \phi) \Rightarrow_{SG} \phi. \quad ■$$

This means that in order to derive a property it is enough to derive it assuming that either the next state transition will be witnessed by the SU, i.e. that the SU will be engaged in the state transition through one of its actions (first set of premisses) or that a chosen set of local terms will be kept invariant (last premiss). Any set of local terms will do.

Definition (Induction Rule).
In order to prove the temporal properties we use the following induction rule:

$$(\phi \rightarrow \mathbf{X}\phi),\ (\boxed{\text{BEG}} \rightarrow \phi) \Rightarrow_{SG} \phi,$$

which should be derivable from the temporal axiomatization. ■

This rule states that we may infer a property ϕ from the fact that it is a temporal invariant and that it holds in the initial state. Note that $\Rightarrow_\theta$ gives consequence "in a model", i.e. the premisses are assumed to be satisfied at any possible instant and we derive the satisfaction of the conclusion at any possible instant.

3.4 The Specification of Systems of Interconnected ADVs and ADOs

Until now we have presented a formal framework in which system units can be specified and enphasized the role of the encapsulation and open semantics principles in enabling only properties to be derived which hold independently of the environments in which the components will be placed.

In this section we describe how to specify systems of interconnected ADVs and ADOs. The structure of the system is explicitly recorded in its specification and the inference mechanisms of the logic are extended in order to deal with this new information. The basic notion in this context is the notion of sub-unit or component. The corresponding component-of relationship is formalized through the notion of signature morphism. A signature morphism is a mapping from the signature of the component to the signature of the composite (system) that contains it. The idea is to use the two primitives of the underlying logical theory, i.e. morphisms and pushouts. In fact, we use the notion of morphisms (specifically description morphisms) to formalize the "composition-of" relationship. We also use colimits in order to obtain the ADV resulting from a system specification of interconnected ADVs and ADOs through "middle theories" as we shall see later in this section. However, colimits can be seen as a sequence of pushouts.

Definition (Morphism Between SU Signatures).
Given two SU signatures $SG_1 =< DT_1, AT_1, AC_1 >$ and $SG_2 =< DT_2, AT_2, AC_2 >$, a morphism σ from SG_1 to SG_2 consists of:

1. An usual morphism of ADT signatures $\sigma_v : DT_1 \longrightarrow DT_2$;

2. For each $f : s_1, \ldots, s_n \longrightarrow s$ in AT_1, there is an attribute symbol $\sigma_\alpha(f) : \sigma_v(s_1), \ldots, \sigma_v(s_n) \longrightarrow \sigma_v(s)$ in AT_2;

3. This case depends if SG_1 and SG_2 are ADV or ADO signatures.

   (a) SG_1 and SG_2 are both ADO signatures.
   For each $g : s_1, \ldots, s_n$ in AC_1, there is an action symbol $\sigma_\gamma(g) : \sigma_v(s_1), \ldots, \sigma_v(s_n)$ in AC_2.

   (b) SG_1 and SG_2 are an ADO and an ADV signature, respectively.
   For each $g : s_1, \ldots, s_n$ in $AC_{Q,1}$ ($AC_{E,1}$), there is an action symbol $\sigma_\gamma(g) : \sigma_v(s_1), \ldots, \sigma_v(s_n)$ in $AC_{Q,2}$ ($AC_{C,2}$).

   (c) SG_1 and SG_2 are both ADV signatures.
   For each $g : s_1, \ldots, s_n$ in $AC_{C,1}$ ($AC_{Q,1}$, $AC_{E,1}$ or $AC_{O,1}$), there is an action symbol $\sigma_\gamma(g) : \sigma_v(s_1)$, $\ldots, \sigma_v(s_n)$ $AC_{C,2}$ ($AC_{Q,2}$, $AC_{E,2}$ or $AC_{O,2}$, respectively). ■

Note that in item (b) the effectual actions of the SG_1 (an ADO signature) are associated through the morphism to a subset of the causal actions of the SG_2 (an ADV signature).

The notion of reduct of interpretation structures along a signature morphism is defined as follows:

Definition (Reduct).
Given two SG signatures $SG_1 =< DT_1, AT_1, AC_1 >$ and $SG_2 =< DT_2, AT_2, AC_2 >$, and a morphism σ from SG_1 to SG_2, we define for every SG_2-interpretation structure $\mathcal{SG} =< \mathcal{DT}, \mathcal{AT}, \mathcal{AC} >$ its reduct along σ as the SG_1-interpretation structure $\mathcal{SG}\mid_\sigma =< \mathcal{DT}\mid_\sigma, \mathcal{AT}\mid_\sigma, \mathcal{AC}\mid_\sigma >$ where:

1. For every $s \in S_1$, $s_{\mathcal{DT}\mid_\sigma} = \sigma_{\mathcal{DT}}(s)_{\mathcal{DT}}$;

2. For every $f : s_1, \ldots, s_n \longrightarrow s$ in F_1, $f_{\mathcal{DT}\mid_\sigma} = \sigma_v(f)_{\mathcal{DT}}$;

3. For every $f : s_1, \ldots, s_n \longrightarrow s$ in AT_1, $\mathcal{AT}\mid_\sigma (f)(i) = \mathcal{AT}(\sigma_\alpha(f))(i)$;

4. For every $g : s_1, \ldots, s_n$ in AC_1, $\mathcal{AC}\mid_\sigma (g) = \mathcal{AC}(\sigma_\gamma(g))$. ■

Definition (Abbreviation of the Locality Axiom).
We abbreviate the translation of the locality axiom, i.e. the θ_2 formula

$$\sigma\,(\,(\bigvee_{g \in AC_1} (\exists x_g) g(x_g)) \vee (\bigwedge_{a \in AT_1} (\forall x_a)(\mathbf{X}a(x_a) = a(x_a))))$$

by $SG_1 \longrightarrow^\sigma SG_2$, i.e. by the signature morphism itself. ■

Definition (Morphism of Theory Presentations).
Given the SU descriptions $< SG_1, AX_1 >$ and $< SG_2, AX_2 >$, a morphism $\sigma : < SG_1, AX_1 > \longrightarrow < SG_2, AX_2 >$ is a signature morphism $\sigma : SG_1 \longrightarrow SG_2$ such that:

1. $(AX_2 \Rightarrow_{SG_2} \sigma(\phi))$ is valid for every $\phi \in AX_1$;

2. $(AX_2 \Rightarrow_{SG_2} SG_1 \longrightarrow^\sigma SG_2)$ is valid. ■

This means that given the theory presentations $< SG_1, AX_1 >$ and $< SG_2, AX_2 >$ and a morphism σ between their signatures, besides requiring that the axioms of $< SG_1, AX_1 >$ be translated to theorems of $< SG_2, AX_2 >$,

i.e. that $(AX_2 \Rightarrow_{SG_2} \sigma(\phi))$ be valid for every $\phi \in AX_1$, we have to require that the translation of the locality axiom also be a theorem of $< SG_2, AX_2 >$ for σ to be a morphism of theory presentations (interpretation between theories). That is, we require that the locality of the "smaller" component be preserved. In other words, for a signature morphism to define a component of a system unit it is necessary that the reduct of every model of the component be a locus of the unit. In fact, more than just a locus, the reduct must be a model of the unit in the sense that it validates its axioms.

This results in the inference rule:

Proposition
Given a morphism $\sigma : < SG_1, AX_1 > \longrightarrow < SG_2, AX_2 >$ from $(AX_1 \Rightarrow_{SG_1} \phi)$ we may infer $(AX_2 \Rightarrow_{SG_2} \sigma(\phi))$. ■

This rule allows us to "export" properties of a specification along a morphism, i.e. if a property ϕ can be locally derived from AX_1 in SG_1, then the translation of ϕ under the morphism σ can be locally derived from AX_2 (the axioms of the composite) in SG_2.

Besides being used to formalize the component-of relationship, morphisms are used to express interconnections between system components: two components can interact, for example, by sharing other component, i.e. by having a commom sub-component in which they synchronize ("middle theories"). This form of interconnection is also called aggregation.

Encapsulation means that we cannot separate the attributes (the state) from the actions that update them and, hence, imposes a discipline in the way we can interconnect components (draw morphisms between descriptions).

Furthermore, intuitively, each action symbol in the signature of an object provides a port to which another object may be linked for communication (the parameters of the action symbol provide for the data that are to be exchanged). This port is fixed in the sense that every other object connected to that port must synchronize in order to communicate. Hence, if we want two objects to communicate independently with a third one, each via the same operation, we must make sure that they are assigned different ports.

3.5 Global Reasoning About ADV Systems

As we have done when we have described the local reasoning process, instead of using the translation of the locality axioms, we may use a derived inference rule:

Definition (Global Rule).
Let $\sigma : SG_1 \longrightarrow SG_2$ be a morphism between SG signatures. Let ϕ be a SG_2-formula and $t_1, \ldots, t_n$ be local terms of SG_1. Then,

$$SG_1 \longrightarrow^{\sigma} SG_2,\ \{\ (\sigma(g(x_g)) \rightarrow \phi)\ \mid\ g \in AC_1\ \},\ (\ \bigvee_{1 \leq i \leq n} \sigma(\mathbf{X}t_i = t_i) \rightarrow \phi)\ \ \Rightarrow\ \ \phi.\ \blacksquare$$

That is, we may infer any property over SG_2 by providing that ϕ holds when a designated set of local terms imported from SG_1 is invariant. This is because the morphism preserves locality: only actions imported from SG_1 may change the attributes imported from SG_1. Hence, we can apply this rule along different morphisms to derive properties that can be related in order to obtain properties of the global description.

In summary, the verification process is modular in the sense that it can use the structure of the specification (as given through the morphisms) to import the relevant results from the components as lemmas which can be composed to prove properties of the global system. Hence, it is not necessary to work within the global specification of the system. We can, for example, select the relevant fragment of the specification in order to prove a property that does not involve all the constituents of the system specification.

4 The Semantics of the ADV Specification Constructors

In this section we present the ADV basic structuring modes and indicate how to describe the semantics of the ADV specification constructors from the perspective of each of these structuring modes.

4.1 The Basic Specification Structuring Modes

In this section we present three ways of interconnecting or structuring ADV/ADO systems. The first way is called strongly coupled interconnection, the second loosely coupled interconnection, and the third interconnection by "architectural" pairs of ADV/ADO descriptions.

The first one, that we call strongly coupled interconnection, is based on the notion of description morphism and is a means of formalizing the notion of "component-of" relationship between ADVs (or ADOs). This description morphism, as we have previously described, is a mapping from the description of the ADV component to the description of the ADV that contains it (the composite ADV or a system of interconnected ADVs). In other words, this morphism establishes which attribute and action symbols of the system correspond to attribute and action symbols of the component. The reason why we call this structuring mode strong is that the component is considered in this case as "part" of the composite. The second way is through "middle theories", i.e. in this case we introduce another ADV (object) which specifies what attributes and actions are going to be shared by the two (or more) ADVs that are being interconnected. The third way is to considered "architectural" descriptions of pairs ADV/ADO. In this case we can have a pair ADV/ADO described by design structures or by a loosely coupled interconnection. Design structures consist of a "kernel", a "body" and "interfaces" and specify how a "view" can be associated to an "application" through a "body". In what follows we detail these specification structuring modes.

In the first case, we have strongly coupled interconnection. In this case description morphisms are used to describe the interconnection of ADVs, thus formalizing the "composition-of" relationship. Thus, for example, we can specify that ADV_1 is a component of ADV by the diagram $ADV_1 \longrightarrow^{\sigma} ADV$, where σ is a description morphism. This form of interconnection was formalized in section 3.4 through the notion of theory presentation morphisms. In this case we want an ADV with multiple components of ADV_i, we have $ADV_i \longrightarrow^{\sigma_i} ADV$ $(i = 1, \ldots, n)$.

In the second case, we have loosely coupled interconnection. In this case we use morphisms and "middle theories" to describe the interconnection of ADVs (or ADOs), thus formalizing the "aggregation" relationship. Thus, for example, we can specify that ADV_1 is aggregated to ADV_2 through a middle theory MT by the diagram $ADV_1 \longleftarrow^{\sigma_1} MT \longrightarrow^{\sigma_2} ADV_2$, where σ_1 and σ_2 are description morphisms. We have already indicated how morphisms can be used to express this form of interconnection (also called aggregation) in section 3.4. The middle theory MT is used basically to identify sorts, functions, attributes and actions of two or more system units. In this case we want multiple component ADV_is to be aggregated to an ADV, we have $ADV_i \longleftarrow^{\sigma_i} MT \longrightarrow^{\sigma_2} ADV$ $(i = 1, \ldots, n)$.

The third case refers to how we can build pairs ADV/ADO and how we can use these pairs as the main building blocks in the specification process. In this case we can describe the semantics of the relationship between an ADV and an ADO either through a "desigh structure" or a loosely coupled interconnection. A design structure can be represented by a diagram $ADV \longrightarrow^{\sigma_1} Body\_ADV \longleftarrow^{\sigma_2}$, where the ADV is the view, the ADO is the application, and the $Body\_ADV$ describes how the ADV and the ADO are related. In fact, the $Body\_ADV$ description merely specializes the existing behavior described by the application, which implies that the extension is conservative in the sense that no new properties are added [44]. Furthermore, we hide in the ADV the action symbols that were used by the application, once they have been specialized. A loosely coupled interconnection of an ADV with an ADO can be characterized as we described in the second case. The ADV/ADO relationship is then represented by the diagram $ADV \longleftarrow^{\sigma_1} MT \longrightarrow^{\sigma_2} ADO$.

Let us now explain, for example, how to interconnect two ADVs through a loosely coupled interconnection. This way of interconnecting ADVs can be seen as a means to specify how ADVs interact with each other. An ADV can interact with another ADV by defining channels between them. Note that the channels are themselves objects. As an example, assume that we have two ADVs defined as:

ADV_1

Data Signature:
sorts a
operations $f, g : a$
Attribute Symbols:
$h, i : a$
Action Symbols:
$\alpha_1, \alpha_2, \alpha_3$
Axioms:
$\phi_1, \phi_2, \ldots, \phi_m$

ADV_2
Data Signature:
sorts a
operations $f', g' : a$
Attribute Symbols:
$j : a$
Action Symbols:
α'_1, α'_2
Axioms:
$\phi'_1, \ldots, \phi'_n$

The channel can be defined, for example, as a "middle theory" MT:

MT
Data Signature:
sorts a
Attribute Symbols: $\emptyset$
Action Symbols:
$p1, p2$
Axioms: $\emptyset$

The description of this "middle theory" MT_{ADV_1, ADV_2} contains two action symbols accounting for the two channels (α'_1, α'_2) and includes the sort a in order to express the fact that the ADV_1 and the ADV_2 will use the same version of the sort a. This SU description may be seen as the "cable" (with two "wires") that we shall use to connect ADV_1 and ADV_2.

We have two morphisms (injections) that define the interactions (in which id is the identity morphism):

$\sigma_1 : MT \longrightarrow ADV_1$
Data: id_a
Attributes: $\emptyset$
Actions:
$p1 \longrightarrow \alpha_1$
$p2 \longrightarrow \alpha_2$

$\sigma_2 : MT \longrightarrow ADV_2$
Data: id_a
Attributes: $\emptyset$
Actions:
$p1 \longrightarrow \alpha'_1$
$p2 \longrightarrow \alpha'_2$

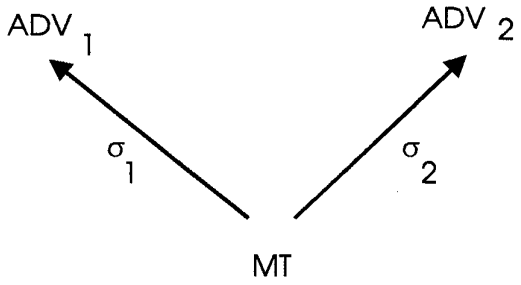


Figure 2: The Interaction Between ADVs

This means that the actions α_1 and α_2 of the ADV_1 are associated with the actions α_1' and α_2' of the ADV_2, respectively. The interaction between the two ADVs is expressed through the diagram in Figure 2.

The joint behavior of these ADVs can be given in terms of the colimit in the category of theory presentations of the diagram above. The colimit of this diagram returns a new ADV description: the description of the "minimal" system unit that admits the depicted ones as components, respecting their interaction as given by the morphisms [24, 25]. In other words, it is the aggregation of the two ADVs (with their shared elements) considered as a system unit itself.

In this context the categorical approach provides us with a faithful notion of module in the sense that we can build complex modules out of simpler ones. Furthermore, the aggregation of system units is another system unit that can be used as a module for building yet more complex objects.

5 Related Topics

In this section we briefly describe related topics which are also being investigated in the context of our research on Abstract Data Views.

Tools and Notation: We expect one result of our research into ADVs to be a software environment to support cooperative design and implementation using the ADV/ADO approach. The various components of the environment will be based on the underlying theory that our research group is currently investigating. It is still premature to produce an entire environment, but we are investigating tools and notations that we know will be useful in this environment. The current theory of ADVs has concentrated on the use of ADVs as user interfaces, and the supporting textual and graphical notations have been directed toward that goal. Investigations reported in [15] indicate that ADVs can be used to characterize interfaces between other media and ADOs, and between ADOs. We are investigating modifications to the notations in order to express these new relationships.

Design Tools: The current textual and graphical formalisms associated with the Abstract Data View model are manipulated manually. Computer-based tools are required if we expect designers to use these formalisms to support the design activity. Specialized systems to draw and annotate ADVcharts, transform ADVcharts into equivalent textual formalisms, animate designs, and check consistency properties are essential tools.

We also intend to develop tools based on the theory described earlier in this proposal to assist the designer in reasoning about properties of interactive system designs such as invariance, response and precedence [35, 37]. Working on these design tools is going on related to the use of the Talisman software engineering environment [38].

Implementation Tools: The bstract Data View design model can be realized using many different software implementation strategies. We intend to examine and categorize these different strategies and produce tools, both for direct implementation and to assist with the transformation of designs into programs. For example, the VX·REXX programming

system [1], which was originally developed as a prototype in our research group, is almost a direct implementation of the Abstract Data View model for one specific programming environment. VX·REXX allows the designer/programmer to translate a design directly into an operating program, thus simplifying many of the steps in the software development cycle. Other programming environments are being examined to determine if it is practical to implement similar strategies.

Design Methods — Case Studies: The value of a software design model, such as the Abstract Data View Model, should be measured in terms of its ability to handle the design of significant software systems. In this regard, we have already performed a number of software design and implementation case studies. We intend to continue further case studies in which we will design new applications and also redesign and reengineer existing software systems. For example, we will investigate the redesign of the Gemini electronic mail package [18], the user interfaces to the JANET [12] class of local area networks, and the design of an editor to support cooperative work on linear graphs. We expect to use many of the case studies as the base for a design approach based on Abstract Data Views.

There are a number of software architectures, particularly in the area of user interfaces, which emphasize separation of concerns, that is, placing specific functions within designated modules or objects. Although these architectures describe the separation of concerns, they generally do not present a method of achieving this separation. Because the Abstract Data View Model and its accompanying formalisms both isolate the user interaction modes from the application and support composition, we believe they provide a method which will direct the designer to designs that maintain the separation of concerns. One case study [43, 8] that has been completed, supports this view. We will continue to investigate this support of separation through the design case studies. Studying the separation of concerns using this model should lead to a more formal approach for identifying and classifying reusable components.

We also are going to investigate how existing design methods such as Structured Analysis [19, 39, 48] and Design [45, 49], JSD [32], Object-Oriented Analysis[9] and Design [5, 47], and Rumbaugh [42] can be used in conjunction with the ADV/ADO approach. All these methods have brought some rigour to the process of creating, and maintaining software systems for a broad variety of applications. We intend to extend the rigour of these and other methods by developing a system that helps the designer formally document steps in whichever design method is being used. Such a system should provide a formal basis for many of the reification methods that are needed to produce an implementation from a design specification.

6 Conclusions and Future Work

This paper has described a logical semantics for Abstract Data Views (ADVs). The semantics for ADVs were given in terms of logical object descriptions ([24, 25]). Lack of space prevented us from presenting the detailed semantics of all the specification constructors used in the theory of ADVs. We are currently working on the detailed semantics for all the ADV specification constructors. We are also working on modularization theorems for the reuse of ADOs interpreted through ADVs (as the one described in section 2) in the context of this logical theory. Another interesting future work is the formal description of the reasoning process related to ADVs through a deductive calculus and the formalization of ADVs as a general module interconnection concept by using the presented approach. This result is related to the formalization of "architectural models" for systems of interconnected components in the context of a logical approach for objects.

References

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

[2] P. Alencar, L. Carneiro-Coffin, D. D. Cowan, and C. Lucena. The Semantics of Abstract Data Views: A Design Concept to Support Reuse-in-the-Large. In *Proceedings of the Colloquium on Object-Orientation in Databases and Software Engineering (to appear)*, May 1994.

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

[4] H. Barringer. The Use of Temporal Logic in the Compositional Specification of Concurrent Systems. In A. Galton, editor, *Temporal Logic and Their Applications*. Academic Press, 1987.

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

[6] L. M. F. Carneiro, D. D. Cowan, and C. J. P. Lucena. ADVcharts: a Visual Formalism for Interactive Systems - position paper. In *SIGCHI Bulletin*, pages 74-77, 1993.

[7] L. M. F. Carneiro, D. D. Cowan, and C. J. P. Lucena. ADVcharts: a Visual Formalism for Interactive Systems. In *Proceedings of the York Workshop on Formal Methods for Interactive Systems (to appear)*. Springer, 1994.

[8] L. Carneiro-Coffin, D. D. Cowan, C. Lucena, and D. Smith. An Experience Using JASMINUM — Formalization Assisting with the Design of User Interfaces. In *Proceedings of the Workshop on Research Issues between Software Engineering and Human-Computer Interaction (to appear)*, Sorrento, May 1994.

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

[10] 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.

[11] 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.

[12] D. D. Cowan, S. L. Fenton, J. W. Graham, and T. M. Stepien. Networks for Education at the University of Waterloo. *Computer Networks and ISDN Systems*, 15:313-327, 1988.

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

[14] 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.

[15] 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.

[16] 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.

[17] D. D. Cowan, E. W. Mackie, G. M. Pianosi, and G. d. V. Smit. Rita - An Editor and User Interface for Manipulating Structured Documents. *Electronic Publishing, Origination, Dissemination and Design*, 4(3):125-150, September 1991.

[18] D. D. Cowan and T. M. Stepien. Electronic Communication and Distance Education. In *Supplementary Proceedings of the Fourth International Conference on Computers and Learning*, pages 14-16, 1992.

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

[20] E. H. Durr. *VDM++ Language Reference Manual*. Afrodite Project Partners, afro/cg/ed/lrm/vs edition, July 1993.

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

[22] J. Fiadeiro and T. Maibaum. Towards Object Calculi. Technical report, Imperial College of Science and Technology, University of London, London, 1990.

[23] J. Fiadeiro and T. Maibaum. *Describing, Structuring, and Implementing Objects*, volume 489 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[24] J. Fiadeiro and T. Maibaum. Temporal Theories as Modularization Units for Concurrent System Specification. *Formal Aspects of Computing*, 4(4), 1992.

[25] J. Fiadeiro and T. Maibaum. Verifying for Reuse: Foundations of Object-oriented System Verification. Technical report, Imperial College of Science and Technology, University of London, London, 1993.

[26] J. Fiadeiro, C. Sernadas, T. Maibaum, and G. Saake. Proof-theoretic Semantics of Object-oriented Specification Constructs. In R. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction (DS-4) - Proceedings of the IFIP TC2 WG 2.6 Working Conference on Object-Oriented Databases - Analysis, Design and Construction*. North-Holland, 1991.

[27] J. Goguen and R. Burstall. *Introducing Institutions*, volume 164 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

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

[29] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *OOPSLA'90*, pages 169-180, 1990.

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

[31] J. Hooman, M. Ramesh, and W. Roever. A Compositional Axiomatization of Statecharts. *Theoretical Computer Science*, 101:289-335, 1992.

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

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

[34] 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.

[35] Z. Manna and A. Pnueli. Tools and Rules for the Practicing Verifier. Technical Report STAN-CS-90-1321, Stanford University, July 1990.

[36] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.

[37] K. Narayana and S. Dharap. Invariant Properties in a Dialog System. In *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development*, May 1990.

[38] L. C. M. Nova, A. V. Staa, D. D. Cowan, and C. J. P. Lucena. *On The Automation of Code Generation for User Interface Models*. submitted to CASCON'94, 1994.

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

[40] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science, IEEE*, pages 45-57, 1977.

[41] 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.

[42] J. Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

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

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

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

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

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

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

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