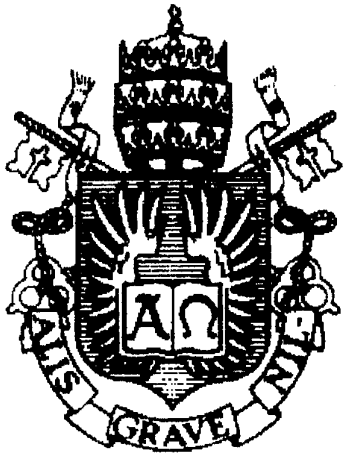

PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 04/95

A Formal Approach for Reusable Objects

Paulo S. C. Alencar
Donald D. Cowan
Carlos José Pereira de Lucena
L. C. M. Nova

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° 04/95

Editor: Carlos J. P. Lucena February, 1995

A Formal Approach for Reusable Objects *

Paulo S. C. Alencar

Donald D. Cowan

Carlos José Pereira de Lucena

L . C. M. Nova

* This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

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

A Formal Approach for Reusable Interface Objects

P.S.C. Alencar *
Computer Science Departament
Computer Systems Group, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
e-mail: alencar@csg.uwaterloo.ca

D.D. Cowan †
Computer Science Departament
Computer Systems Group, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
e-mail: dcowan@csg.uwaterloo.ca

C.J.P. Lucena ‡
Computer Science Departament
Computer Systems Group, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
e-mail: lucena@csg.uwaterloo.ca

L.C.M. Nova §
Computer Science Departament
Computer Systems Group, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
e-mail: luisnova@csg.uwaterloo.ca

PUC-RioInf.MCC

April 5, 1995

Abstract: In this paper we present a formal approach of a new object-oriented design concept to support reuse-in-the-large called Abstract Data Views (ADVs). The ADV approach was created to specify clearly and formally the separation of interfaces from the application components of a software system. Such approach should lead to a high degree of reuse of designs for both interface and application components.

Our specification framework is based on descriptive schemas for both ADVs and ADOs, that should be used as the basic blocks for the system specification, design, and implementation within the ADV approach. These schemas describe the structural, static, and dynamic features of each system object, allowing also the specification of the concurrent aspects of the several system components. Additionally, such schemas can be seen as an underlying structure to support the development of a specification language that describes the interconnection capabilities between interface and application components.

Keywords: Concurrent Programming, Object-Oriented Programming, Formal Specification, Interfaces.

Resumo: Neste artigo nós apresentamos uma abordagem formal para um novo conceito de design orientado a objetos que suporta reuso em ponto grande chamado Visões Abstratas de Dados (ADVs). A abordagem ADV foi criada para especificar clara e formalmente a separação entre os componentes da interface e da aplicação de um sistema de software. Tal abordagem permite um alto grau de reuso de designs tanto para componentes da interface quanto da aplicação.

Nossa abordagem de especificação é baseada em esquemas descritivos para ADVs e ADOs, que devem ser usados como os blocos básicos para a especificação, o design e a implementação segundo a abordagem ADV. Estes esquemas descrevem as características estruturais, estáticas e dinâmicas de cada objeto do sistema e permitem também a especificação de aspectos concorrentes dos vários componentes do sistema. Além disso, tais esquemas podem ser vistos como uma estrutura subjacente para o suporte do desenvolvimento de uma linguagem de especificação que descreve a interconexão entre componentes da interface e da aplicação.

Palavras-chave: Programação Concorrente, Programação Orientada a Objetos, Especificação Formal, Interfaces.

---

*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, Brasil. Email: alencar@csg.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, Brasil. Email: lucena@csg.uwaterloo.ca

§L.C.M. Nova is a PhD candidate in the Computer Science Department at University of Waterloo and holds a doctoral fellowship from CNPq. Email: lcmnova@neumann.uwaterloo.ca

1 Introduction

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 or functions 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.

The *Abstract Data View* [14] approach uses an object-oriented formal design model which supports separation of concerns and reuse of design specifications. The basic constructs of the ADV approach are the Abstract Data View (ADV) and the Abstract Data Object (ADO), which represent, respectively, interface objects (views and interactions) and application objects which are independent of the interface. These objects provide a disciplined approach to design which supports separation of concerns, and should lead to a wide and consistent reuse of design specifications for both interface and application components.

Various architectural models and programming approaches [5, 30, 23, 24, 4, 10, 18, 28, 26, 31, 34, 27, 36] have been proposed that clearly separate the user interface and its corresponding application. However, with the architectural models, little guidance is given on designing a program to have a reasonable level of assurance that the architecture will be followed. Specific implementation techniques such as MVC [31] and ALV [28] have also been reported in the literature. These rely on contemporary programming models and have been illustrated by several examples. For example, MVC was originally used in Smalltalk and ALV used constraint programming in a LISP environment. Although these are excellent implementation models, it is often difficult to map these strategies into other programming paradigms. The introduction of structures and operators to support separation of interface and application in the design is one of the major contributions of the ADV design model.

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) [32], to support concurrency in a cooperative drawing tool, and to design and implement both a ray-tracer in a distributed environment [38] and a scientific visualization system for the Riemann problem. The VX-REXX [42] system that was built

as a research prototype, was motivated by the idea of composing applications in the ADV/ADO 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 [41]. In addition, we have shown in [12] 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 [37] and megaprogramming [43].

2 Concepts of the ADV model

In general, an ADV may be considered as an specialization of an object with characteristics to support the development of general interfaces. As a consequence, the ADV model will contain most of the characteristics that are inherent in the theory of objects [35].

Both ADVs and ADOs are objects and are composed of data signatures, attributes, and actions. ADVs are modified ADOs that support the role of interfaces in 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 capabilities of ADOs and ADVs. An ADO has no knowledge of its interfaces (ADVs), thus ensuring independence of the application from its interface. On the other hand, an ADV does know about its associated ADO and can query the state of it by means of a mapping between both. Further details will show the importance that this asymmetry will have within the model.

2.1 Actions

Before considering the concepts associated with ADVs and ADOs, we introduce here some basic concepts about actions. We call *actions* the programming functions and input events that act on an object to change or query its state. According to its origin, actions can be distinguished into two categories: causal actions and effectual actions.

We use the term causal actions to denote the input events occurring in an ADV. Causal actions are triggered from outside the system and internal objects cannot generate this kind of action. For example, a key stroke or a mouse click are typical input events that characterize a causal action. Effectual actions are the actions generated directly or indirectly by a causal action. Figure 1 shows the possible dependencies between the two types of actions, while Figure 2 shows a relationship illustration of action occurrences.

$$Action_{causal} \longrightarrow Action_{effectual} \qquad Action_{causal} \not\longrightarrow Action_{causal}$$
$$Action_{effectual} \longrightarrow Action_{effectual} \qquad Action_{effectual} \not\longrightarrow Action_{causal}$$

Figure 1: Action dependencies.

A causal action may originate zero, one, or several effectual actions. On the other hand, an effectual action cannot generate a causal action. This means that if we visualize a tree of related

actions occurring over time, then a causal action must appear only at the root of a tree of actions. This syntactic rule also characterizes the ADV-ADO interaction.

Considering that internal elements should not trigger any event external to the system, and in addition, internal operations or events should be a result of external stimuli, the action dependencies introduced in this section are quite intuitive. Moreover, these dependency definitions lead us to the conclusion that only causal actions (input events) can change the state of the system. The notion of actions is used in further sections with respect to interconnection between ADVs and ADOs.

2.2 Abstract Data Objects

An Abstract Data Object (ADO) is an object in that it has a state and a public interface that can be used to query or change this state. Every action of the public interface of an ADO should be an effectual action and can only be triggered as a consequence of an ADO or an ADV action. The action of external events such as input events or user commands do not directly act on an ADO, thus conserving the notion of an interface disconnected from an ADO.

In the ADV design model, we extend the concept of ADO to allow composition of ADOs through specification constructors, where composition implies that ADOs are composed of a number of constituent ADOs and that one ADO encloses its constituents. In using composition we also assume that the enclosing ADO knows the identity of its constituents, but the enclosed ADOs do not know the identity of the enclosing ADO.

Specification constructors for inheritance, sets, and sequences are also part of the formalism. They are described in [1, 2, 3] and all of these constructors are composition-based, which means that they can be interpreted through a primitive composition specification construct.

2.3 Abstract Data Views

ADV is a design structure conceived to act as general user interface and to achieve a separation of concerns by providing a clear separation at the design level between the application and its interface. ADVs are extensions to ADOs to support the design of user and module interfaces. Since they are ADO extensions, ADVs also support the specification constructors described in the previous section.

As shown in Figure 2, the interaction between an ADV and a user is by means of input events and output messages. The action interface of an ADV is invoked by input events, that is, an ADV action can be triggered by operations such as a mouse click, a keyboard event, or a timer. Thus, the action interface of the ADV extends the ADO interface to include external or causal events. ADVs also send output actions such as display commands, if their appearance is altered through an input message or a change in the state of an associated ADO.

Since the ADO has no knowledge about its associated ADVs, every interaction between an ADV and an ADO has an ADV as origin point. Such interactions have the purpose of either changing or querying the state of the ADO. Changes in the ADO state are the result of an effectual action, which is inside this ADO, triggered by another action. Queries about the ADO state are specified by a direct mapping from the ADV to the ADO, which is shown in Figure 2.

The mapping is a linking mechanism established between the *owner* variable of the ADV and the query functions inside the ADO public interface. Considering that this mapping is a design

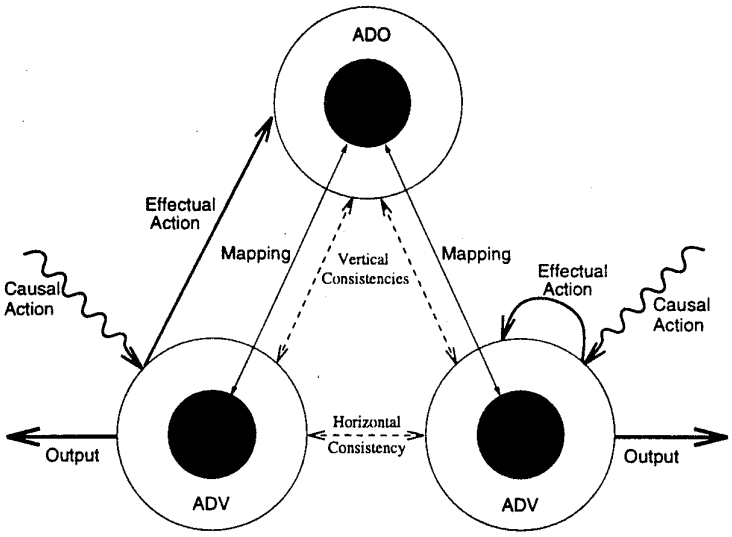

Figure 2: An ADV/ADO interaction model.

concept, the query functions inside the ADO public interface can also be modeled as actions, even if they do not change the state of the ADO as "normal" actions do. This is possible because we can have the result values of a query stored in the parameters of an action, as we will see later.

To illustrate the concepts introduced in this section, we describe the simplified behavior of a chess game prototype as a response to an external stimuli. If the user commands are activated by a mouse, a click on a chess piece would generate an input event for the *piece* ADV. The input event (causal action) triggers effectual actions in order to change the state of the associated *game* ADO. While the *game* ADO is changing its state as a response to the user move, the *board* ADV monitors the ADO state through the mapping mechanism. When an internal operation in the *game* ADO provokes a change in the state of this ADO, the *board* ADV detects this change through the mapping and, if appropriate, sends an output message that updates the user view.

2.4 Interconnection

Communication between an ADV and an ADO instance is essentially a synchronous invocation of actions that are mapped between the ADV and ADO instances, as described in the previous section. The synchronous approach contrasts with other user interface models, where a component must handle both synchronous and asynchronous invocations from other components. Handling asynchronous invocations is considerably more complex than handling synchronous invocations, since it requires error-prone mechanisms such as signals, interrupts, or callbacks. With an explicit mapping we enforce a one-way communication and, as a consequence, have fewer interconnections, thus ensuring that the role and scope of the interface are defined unambiguously.

Although communication between the ADV and ADO is a synchronous process, we can still model asynchronous interconnections using the ADV approach. This is achieved by using a design structure where a "body ADV" is placed between the ADVs (views) of an ADO and the ADO

itself. This body ADV defines relationships of actions and attributes in between the ADVs and the ADO. Such relationships should illustrate the correspondence of elements in an ADV with its associated ADO. Each relationship is defined for a pair of elements where one element has a correspondence inside an ADV and the other has a correspondence inside the ADO. In general, the body ADV may be regarded as a design element that specifies how the ADVs interact with its associated ADO. Figure 3 illustrates the design structures that model synchronous and asynchronous interconnections.

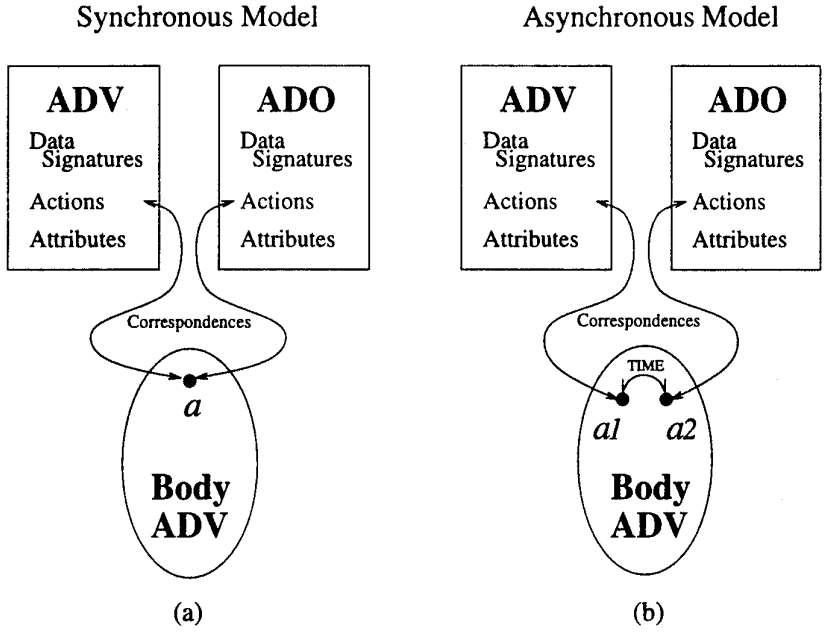

Figure 3: Interconnection models.

Figure 3(a) illustrates the modeling of a synchronous interconnection process, where *a* is a single action inside the body ADV that has a correspondent action in both ADV and ADO. In Figure 3(b) we have the asynchronous action invocation model, where *a1* and *a2* are two actions corresponding to an ADV and an ADO action, respectively. Besides the *a1* and *a2* correspondences, the body ADV should also represent a relationship that introduces asynchrony between the actions. For example, a delay in invocation time.

Although we have been using in this section the structure of a body ADV to describe how to interconnect many ADVs with an ADO, we can also use this same structure to describe interconnections between only ADVs or only ADOs. In [39, 15, 19, 13] there are some interesting applications where such design structures might be very useful.

2.5 Consistency

The separation between interface and application makes it possible to create different visual representations for a single collection of ADOs. For example, the user interface for a clock ADO could

be represented in a digital view, an analog view or both. Another possible application for the clock ADO, is to use its attribute values as input to another ADO. In such case, an ADV would be placed as an interface between two ADOs, instead of working as the interface between the user and the ADO.

As an implication of the flexibility introduced by the separation of concerns, consistency should exist among the many visual representations (ADVs) of a single ADO and the ADO itself. The consistency among the different ADVs is called *horizontal consistency.* The one between the visual object (ADV) and its associated ADO is called *vertical consistency.* These consistency properties must be guaranteed by the specification of ADVs, ADOs, and the environment that holds them. Figure 2 illustrates the above concepts using a model of an ADO containing two distinct interfaces.

Using again the clock example, we might say that horizontal consistency will guarantee that every different view associated with the clock ADO will be showing the same time. On the other hand, vertical consistency will make sure that a view is illustrating the same time specified by the clock ADO attributes. The time is provided to the ADVs through the mapping from the ADV to the state of the ADO, since the state of the clock ADO is changed by a timer and not by the user interface.

3 ADV and ADO specifications

In this section we introduce some abstract schemas for the specification of ADVs and ADOs. These abstract schemas are useful tools for both formal and informal program specifications [9, 20]. In fact, our schemas were based on the ones described in [9, 22, 21]. In addition to the syntactic descriptions of ADVs and ADOs, we also describe in the schemas how the objects are created, change states, and destroyed through the use of actions. The definition of a schema for each ADV and ADO in a software system is an important step in the software development process, since they constitute the basic entities of the system. The structures used are be shown to be suitable for refinement operations that should occur at each stage of development.

3.1 Descriptive schemas

ADVs and ADOs have distinct roles in a software system and, as a consequence, they are described by different schemas. These schemas are not the actual objects inside a system, but rather descriptions of their static and dynamic properties and declarations of entities that are used within the scope of the object. Therefore, every ADV or ADO structure is subdivided into three sections: declarations, static properties, and dynamic properties.

Despite having completely different roles in a software system, an ADV is an extension of an ADO, thus conserving most of the properties and general structure that characterizes an ADO. Therefore, we initially introduce the abstract schema for ADVs, and then mention the distinctions between ADV and ADO schemas.

Since every ADV is considered as an interface part of an application ADO, the schema seen in Figure 4, introduces the link *ADV_Name/ADO_Name* through which the ADV is related to the associated ADO. The exception to this rule is when the ADV does not have a corresponding ADO, such as an ADV representing a button that changes the colors of the display screen. In this case, the *ADO_Name* part is left blank.

```
ADV ADV_Name for_ADO ADO_Name
    Declarations
        Data Signatures      - sorts and functions
        Attributes           - observable properties of objects
        Causal Actions       - list of possible input actions
        Effectual Actions    - list of possible effectual actions
        Nested ADVs          - allows composition, inheritance, sets, ...
    Static Properties
        Constraints          - constraints in the attributes values
        Derived Attributes   - non-primitive attribute descriptions
    Dynamic Properties
        Initialization       - initializes attributes
        Interconnection      - describes the communication process among objects
        Valuation            - the effect of events on attributes
        Behavior             - behavioral properties of the ADV
End ADV_Name
```

Figure 4: A descriptive schema for an ADV.

By means of the *data signature* declarations, the signatures of sorts and functions are stated in the ADV. Among sort expressions we may have the basic abstractions, such as *integer, string, etc.* and the application specific abstractions including object instances or user-defined sorts. Sort constructors, such as *set, union, etc.*, can also be applied to compose complex sort expressions, since the semantics involved in a sort signature is an association of a set of values with a sort expression. The functions, denoting operations over given values, are also part of the data signature section.

We use the pair <*sorts, functions*> in our object specifications for establishing the available types and operations on the values that are stored in the attributes. We do this because we want to support (abstract) specification of the space of values as well, and not just the behavior of objects. However, if we assume that all ADVs and ADOs operate over a fixed collection of data types (integers, arrays, etc.) then we do not need that component in a signature.

The *attributes* declaration denotes the state or the set of features of an object that can change over time. Attributes are the state memory of an ADV or an ADO. This means that attributes are the observable properties of an object and are used by other objects to report on the current state of that object. Although the attribute values may change over time, the set of attributes remains unaltered from creation until destruction of the object. However, regarding that sort expressions contain the undefined element $\perp$, the attributes are optional by default.

According to the definitions introduced in Section 2.1, actions occur in both ADVs and ADOs. It is also important to note that attribute values can only be affected by local actions, which are the actions defined in the same schema of the attribute to be changed. In the ADV schema, actions are divided into two groups: *causal actions* and *effectual actions.* The distinction between both types was explained before and, in addition to that, we could subdivide each type of action into two other subtypes: observer actions and changing actions. The observer actions are the actions

whose objective is to query the state of the associated ADO, while the changing actions do modify the state of the system.

In [2] there is a formal description of some approaches to indicate how specification of components can be made reusable. Composition, inheritance, sets and sequences are the nesting mechanisms in the ADV model for combining and reusing specifications. The ADV containing the *nested ADVs* is called a composite object or parent, while the nested ADVs are called component objects. All objects are components of some composite object; objects which do not belong to a composite object are a component of the operating environment or "system". Composite objects are responsible for creating the instance of the components. Since the creation action can happen only once in a life of any object, then a component object can never be nested in two distinct composite objects. The same rules used here for ADVs will also apply to ADOs.

Constraints and *derived attributes* are considered static properties of an object, since their existence does not affect the state of the object. A constraint expression is a mechanism to establish bounds on attribute values. As a consequence, possible object states can be restricted by constraints.

Being another type of static property, a derived attribute is like an attribute in that it is a property of the object itself, and computing it does not change the state of the object [40]. A derived attribute may have its value determined by means of operations over base attribute values only.

The first of the dynamic properties of ADVs to be discussed is the *initialization* process. The initialization of an object is generally defined by the triggering of actions that initialize the attribute values of this object. Before the creation of an object, every attribute is valued as *undefined*. In addition, attributes that are not affected by actions remain unchanged after the occurrence of that action. Since the attribute values may or may not be initialized by the time of object creation, an attribute may hold the undefined value even after the initialization process.

In the *interconnection* section there should be a description of which attributes and actions in one object have a synchronous correspondence with the attributes and actions in any other object. The mapping of attributes of an ADV and its associated ADO is described as an interconnection property, and is generally defined in a body ADV schema. The expressions in this section should also describe all the effects of an action declared in this object. Such effects could be the triggering of actions inside or outside this object. Another important consideration to be mentioned here is that an ADO action cannot trigger an ADV action, since the ADO has no knowledge about the existing ADVs.

All the actions previously declared in the effectual and causal action sections should also have its valuation properties defined in the *valuation* section. The valuation properties of an object describes the changes occurring in attribute values of this object as an immediate consequence of a triggered action. The valuation rules are applied only if all the specified pre-conditions for the occurrence of an action, if any, are satisfied. Both the expression that will determine the new attribute values and the pre-conditions are to be evaluated in the state before the action occurrence.

Behavior description is in general a complex task. Most of the object-oriented models represent system behavior by means of transition networks, which are augmented state machine diagrams [17]. In this paper, we chose a textual representation to describe the object behavior. However, there are a number of graphical representation languages that we could have chosen to describe the

behavior of objects [6, 25, 8, 7].

```
ADO ADO_Name
    Declarations
        Data Signatures      - sorts and functions
        Attributes           - observable properties of objects
        Effectual Actions    - list of possible effectual actions
        Nested ADOs          - allows composition, inheritance, sets, ...
    Static Properties
        Constraints          - constraints in the attributes values
        Derived Attributes   - non-primitive attribute descriptions
    Dynamic Properties
        Initialization       - initializes attributes
        Interconnection      - describes the communication process among objects
        Valuation            - the effect of events on attributes
        Behavior             - behavioral properties of the ADO
End ADO_Name
```

Figure 5: A descriptive schema for an ADO.

In Figure 5 we see the schema of an ADO. Since its characteristics are very similar to the ones in the ADV schema, we do not need to describe the details of every section. The first distinction in the structure is found in the header of the schemas. Since an ADO has no knowledge about its associated ADVs, the header of an ADO schema contains only the name of the ADO (*ADO_Name*).

Another structural difference between ADV and ADO schemas lies in sections related to the description of actions. As mentioned before, an ADO has no definition of causal actions in its structure, thus every ADO action is effectual.

3.2 The specification syntax

In this section we introduce a preliminary formal syntax for the specification schema informally introduced in the previous section. Although we opted to use a VDM-like notation [29] to describe the ADV approach in some other works, here we opted for a notation that could clearly specify all the sections composing the specification schema. Therefore, the specification syntax is presented essentially through a temporal logic formalism [33], and it may be regarded as an initial step towards the specification of a formal semantics for the ADV design approach.

The data signature part of an object consists of a set S of sort names, and a set F of function declarations which have the form $f : s_1, \ldots, s_n \longrightarrow s_0$, with $s_i \in S$ for $i = 0, \ldots, n$, where $s_1, \ldots, s_n$ is called the domain of f, s_0 is called the codomain of f and n is called the arity of f.

The attributes are represented as a set of declarations of the form $g : s_1, \ldots, s_n \longrightarrow s_0$, with $s_i \in S$ for $i = 0, \ldots, n$, where g is an attribute name, $s_1, \ldots, s_n$ $(n \geq 0)$ denote optional parameter sorts of the attribute, and s_0 is a sort determining the range of the attribute.

Actions are represented as a set of declarations of the form $a : s_1, \ldots, s_n$, with $s_i \in S$ for $i = 0, \ldots, n$, where a is an action name, and $s_1, \ldots, s_n$ $(n \geq 0)$ represent the optional parameter sorts of the action. The parameters in the action description may have an important role inside a system specification. They might be used to define the effect of an action occurrence on the current state of an object, or they could be used to describe the transmission of data during the interconnection process. In the first case the parameter values are used to update the attribute values of the object. This process is described in the *valuation section.* In the second case, the parameters may be considered as a mechanism to transmit data from one object to another. Such mechanism is particularly useful to model the response of ADOs to query actions.

In the *Nested ADV (ADO)* section we should introduce the list of all the component objects of a composite ADV or ADO with the specification constructors that support nesting. Therefore, we specify the syntax of this section by the expression $[constr]obj$, where *constr* is a specification constructor that can be specified by the terms **component, inherit, set of** and **sequence of**, and *obj* denotes the nested ADV or ADO name.

The static properties of an object are represented by closed formulas. While constraint formulas refer to properties that must be true at all time instants, derived attribute formulas define the derivation rules for the non-primitive attributes from the primitive (base) attributes. Constraints are represented by the temporal logic expression $\Box\ \phi$, where ϕ is an invariance property, and $\Box$ is the temporal logic operator "always". Derived attributes are also expressed by means of temporal logic formulas.

In order to express the initialization of a system unit (ADV or ADO), we must introduce initialization actions that are to be executed when this object is created. Actually, the initialization actions are effectual actions that are executed in the beginning of the object life time. This is expressed by the formula $\boxed{\text{BEG}} \rightarrow a_1(s_1, \ldots, s_i) \wedge \ldots \wedge a_N(s_1, \ldots, s_j)$, where $a_1, \ldots, a_N$ are initialization actions that must be declared as effectual actions and must have a corresponding valuation entrance to describe how it changes the object attributes.

The interconnection section is described by morphisms of actions and attributes. The morphisms (or mappings) are used to define which attributes and which actions are to be identified between objects. The intended interpretation is that the shared attributes must always have the same values and the shared action must happen simultaneously. The representation of a morphism is specified by expressions of the form $obj\_name1.element \longmapsto obj\_name2.element$ where $obj\_name.element$ refers to attributes or actions which are defined inside the obj_name object and can be seen by the current object. When the obj_name term is missing, the element should be one that is declared inside the current object.

The valuation for an action a is described by means of temporal formulas of the form $a \rightarrow pre\_condition$ and $a \rightarrow att_1 = value_1 \wedge \ldots att_n = value_n$. The first form of temporal formula defines a pre-condition on the occurrence of the action named a. In addition, since a single action may have zero, one, or many pre-conditions, the number of formulas to describe the valuation of an action is unlimited. The other form of temporal formula establishes post-conditions on the occurrence of an action. Such formula shows the effects of an action over the attribute values of the object.

The behavior will be expressed by temporal logic formulas that express the sequencing of the actions and changes in the state of the object. A typical behavior expression is described by

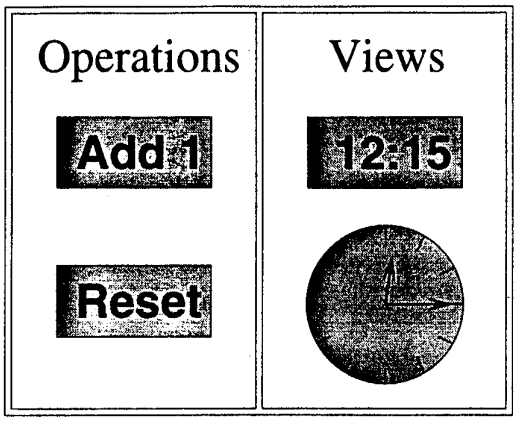

Figure 6: The manual clock interface.

pre_state $\wedge$ *action_name* $\rightarrow$ *post_state*, where *pre_state* and *post_state* are states of the object that are declared in the data signature section as boolean functions, and action_name is an action of the object.

3.3 The manual clock example

In this section we describe the specification of a simple application example using the schemas previously introduced. The example consists on the operation of a clock that is updated by the user. The user interface of this application consists of two buttons that modifies the clock time specified by the attributes inside the clock ADO, and two views of the contents of this ADO: a digital and an analogical clock view. Although simple, the manual clock illustrates most of the concepts introduced in this paper. The clock interface is illustrated in Figure 6.

We will be using some symbols associated to the temporal logic notation. Besides the "always" operator ($\Box$), we also use the operator "next" ($\mathcal{X}$), which allows the denotation of the variable value after the execution of the command. Additionally, we use underlined words (e.g., nat) to express a sort name which we assume to be pre-defined.

We start the specification of the example system by describing the application ADO, as Figure 7 shows. It is important to note that such ADO contains all the specification of the application dynamics and it has no user interface detail embedded in its structure. Consequently, we may consider this ADO specification as potentially reusable.

In figure 8 we define the ADV specification schemas composing the interface. The *Button_Add_1* ADV defines an interface button that recognizes when the user clicks a mouse button onto the region where it is drawn. The *Digital_Clock_View* ADV is an output dispositive that displays to the user the values stored in the application. The *Interface_Window* is a composite ADV responsible for nesting the many interface dispositives into a single interface unit. This nesting capability shows itself in the screen, where the component ADVs are drawn inside its parent region.

For reasons of simplicity, the specification of the *Button_Reset* and *Analogic_Clock_View* ADVs were omitted, since its specifications are very similar to the *Button_Add_1* and *Digital_Clock_View* ADVs, respectively. Additionally, we are also assuming the existence of mappings of attributes and

ADO Clock_Application
Declarations
Attributes
Hours, Minutes: nat;
Effectual Actions
Add_1; Reset; Show(time);
Static Properties
Constraints
$\Box(0 < \text{Hours} \leq 12)$;
$\Box(0 \leq \text{Minutes} < 60)$;
Derived Attributes
Time_in_Minutes = 60× Hours + Minutes;
Dynamic Properties
Initialization
$\boxed{\text{BEG}} \rightarrow$ Reset;
Valuation
$\text{Reset} \rightarrow \mathcal{X}\ \text{Hours} = 12 \wedge \mathcal{X}\ \text{Minutes} = 0$;
$\text{Add\_1} \rightarrow (\text{Minutes} = 59 \rightarrow \mathcal{X}\ \text{Minutes} = 0 \wedge \mathcal{X}\ \text{Hours} = \text{Hours} + 1) \vee$
$(\neg\ (\text{Minutes} = 59) \rightarrow \mathcal{X}\ \text{Minutes} = \text{Minutes} + 1)$;
$\text{Show(time)} \rightarrow \text{time} = \text{Time\_in\_Minutes}$;
End Clock_Application

Figure 7: The ADO for the manual clock application.

actions between the composite ADV and its components.

The *body_ADV* is a design mechanism in which we define how the user interface and the application objects are interconnected, by describing which attributes and actions are shared between these objects. It is important to note that the symbol $\longmapsto$ in the body ADV does not define the order in which actions occur in time. It only identifies attributes and actions of different objects.

4 Conclusion

We have introduced a formal specification framework for the development of reusable objects in the ADV object-oriented design approach. The paper describes the basic aspects of the approach, its issues towards reuse and separation of concerns, and the concepts involving the system units (ADVs and ADOs). The framework is supported by schemas that describe the structural, static, and dynamic features of each system object, allowing also the specification of the concurrent aspects of the several system components.

Although we have used a temporal logic formalism to represent the properties of the ADV and ADO schemas, many other alternative languages might have been used to define the components of the system. This choice was based on a uniform framework suitable for the description of every element composing the schema structure.

This specification model can be used as a basic formal tool for the development of an ADV object-oriented formal design method, since it is flexible enough to represent all the characteristics related to the object, functional, and dynamic models composing a design methodology [40].

ADV Button_Add_1
Declarations
Attributes
Pressed: boolean;
Causal Actions
Press_Button; Release_Button;
Effectual Actions
Initialize_Pressed; Draw_Button(position, name);
Dynamic Properties
Initialization
BEG → Initialize_Pressed;
BEG → Draw_Button(POSITION_1, 'ADD 1');
Valuation
Initialize_Pressed → $\mathcal{X}$ Pressed = *false*;
Press_Button → Pressed = *false*;
Press_Button → $\mathcal{X}$ Pressed = *true*;
Release_Button → Pressed = *true*;
Release_Button → $\mathcal{X}$ Pressed = *false*;
Behavior
Status_Down ∧ Release_Button → Status_Up;
Status_Up ∧ Press_Button → Status_Down;
End Button_Add_1

ADV Digital_Clock_View
Declarations
Attributes
Hours_Display, Minutes_Display: nat;
Effectual Actions
Update_View; Draw_Digital_View(position, hours_display, minutes_display);
Dynamic Properties
Initialization
BEG → Update_View;
Behavior
Status_Updated ∧ Update_View → Status_Updating;
Status_Updating ∧ Draw_Digital_View(POSITION_3, Hours_Display, Minutes_Display) → Status_Updated;
End Digital_Clock_View

ADV Interface_Window for_ADO Clock_Application
Declarations
Attributes
Status: nat;
Effectual Actions
Draw_View(position);
Nested ADVs
Component Button_Add_1, Button_Reset, Digital_Clock_View, Analogic_Clock_View;
Dynamic Properties
Initialization
BEG → Draw_View(POSITION);
End Interface_Window

Figure 8: Some ADV schemas composing the interface.

ADV Body_ADV
Declarations
Attributes
Hours_Link, Minutes_Link: nat;
Effectual Actions
Add_1_Link; Reset_Link;
Dynamic Properties
Interconnection
Hours_Link ⟼ Clock_Application.Hours;
Minutes_Link ⟼ Clock_Application.Minutes;
Add_1_Link ⟼ Clock_Application.Add_1;
Reset_Link ⟼ Clock_Application.Reset;
Add_1_Link ⟼ Button_Add_1.Press_Button;
Reset_Link ⟼ Button_Reset.Press_Button;
Hours_Link ⟼ Digital_Clock_View.Hours_Display;
Minutes_Link ⟼ Digital_Clock_View.Minutes_Display;
Add_1_Link ⟼ Digital_Clock_View.Update_View;
Reset_Link ⟼ Digital_Clock_View.Update_View;
Hours_Link ⟼ Analogic_Clock_View.Hours_Display;
Minutes_Link ⟼ Analogic_Clock_View.Minutes_Display;
Add_1_Link ⟼ Analogic_Clock_View.Update_View;
Reset_Link ⟼ Analogic_Clock_View.Update_View;
End Body_ADV

Figure 9: The ADV Body_ADV schema.

It can also be extended to handle cooperative aspects of distributed and/or concurrent highly-interactive systems. Particularly, the support to asynchronous communication allows us to model the interaction among cooperative (visual) agents throughout the design process.

References

[1] P.S.C. Alencar, L.M.F. Carneiro-Coffin, D. D. Cowan, and C.J.P. 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).* Kluwer Press, May 1994.

[2] P.S.C. Alencar, L.M.F. Carneiro-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.

[3] P.S.C. Alencar, L.M.F. Carneiro-Coffin, D. D. Cowan, and C.J.P. Lucena. Towards a Logical Theory of ADVs. In *Proceedings of the Workshop on the Logical Foundations of Object Oriented Programming ECOOP94 (to appear)*, July 1994.

[4] Len Bass and Joëlle Coutaz. *Developing Software for the User Interface.* The SEI Series in Software Engineering. Addison-Wesley, 1991.

[5] 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.

[6] L.M.F. Carneiro, D.D. Cowan, and C.J.P. Lucena. ADVcharts: a Visual Formalism for Describing Abstract Data Views. Technical Report 93–20, Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, 1993.

[7] George W. Cherry. S-R Machines: a Visual Formalism for Reactive and Interactive Systems. *ACM Software Engineering Notes*, 16(3):52–55, July 1991.

[8] Derek Coleman, Fiona Hayes, and Stephen Bear. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.

[9] Stefan Conrad, Martin Gogolla, and Rudolf Herzig. TROLL light: A core language for specifying objects. Technical Report 92-02, Technische Universitat Braunschweig, December 1992.

[10] Joelle Coutaz and Sandrine Balbo. Applications: A Dimension Space for User Interface Management Systems. In Scott P. Robertson, Gary M. Olson, and Judith S. Olson, editors, *Reaching Through Technology, CHI'91 Conference Proceedings*, pages 27–32, New Orleans, Louisiana, April 27-May2 1991. Addison-Wesley.

[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, 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.

[13] 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 Dept and Computer Systems Group, University of Waterloo, Canada, November 1993.

[14] D.D. Cowan and C.J.P. Lucena. Abstract Data Views: An Interface Specification Concept to Enhance Design. *IEEE Transactions on Software Engineering*, 21(3):229–243, March 1995.

[15] D.D. Cowan, C.J.P. Lucena, and A.B. Potengy. A Programming Approach for Distributed Computing. *Submitted to Distributed Computing*, 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] Dennis de Champeaux, Douglas Lea, and Penelope Faure. *Object-Oriented System Development.* Addison-Wesley, Reading, Massachusetts, 1993.

[18] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design Inc., and SunSoft, Inc. *The Common Object Reqest Broker: Architecture and Specification*, OMG document number 91.12.1, revision 1.1 edition, December 1991.

[19] Donald D. Cowan et al. Program design using abstract data views - an illustrative example. Technical Report 92-54, University of Waterloo, December 1992.

[20] B. Fields, M. Harrison, and P. Wright. From Informal Requirements to Agent-based Specification: an Aircraft Warnings Case Study. Technical report, University of York, August 1993.

[21] M. Gogolla, N. Vlachantonis, R. Herzig, G. Denker, S. Conrad, and H.D. Ehrich. The KORSO approach to the development of reliable information systems. Technical Report 94-06, Technische Universitat Braunschweig, June 1994.

[22] Martin Gogolla, Stefan Conrad, and Rudolf Herzig. Sketching Concepts and Computational Model of TROLL Light. In *Proceedings of Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'93)*, Berlin, Germany, March 1993. Springer.

[23] M. Green. Design Notations and User Interface Management Systems. In Günther E. Pfaff, editor, *User Interface Management Systems–Proceedings of the Workshop on User Interface Management Systems held in Seeheim, FRG, November 1-3, 1983.* Springer-Verlag, 1985.

[24] M. Green. Report on Dialogue Specification Tools. In Günther E. Pfaff, editor, *User Interface Management Systems–Proceedings of the Workshop on User Interface Management Systems held in Seeheim, FRG, November 1-3, 1983.* Springer-Verlag, 1985.

[25] David Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[26] Rex Hartson. User-Interface Management Control and Communication. *IEEE Software*, 26(1):62–70, January 1989.

[27] Ralph D. Hill. Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction–The Sassafras UIMS. *ACM Transactions on Graphics*, 5(3):179–210, July 1986.

[28] 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.

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

[30] D.R. Jr. Olsen. Presentational, Syntactic and Semantic Components of Interactive Dialogue Specifications. In Günther E. Pfaff, editor, *User Interface Management Systems–Proceedings of the Workshop on User Interface Management Systems held in Seeheim, FRG, November 1-3, 1983*. Springer-Verlag, 1985.

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

[32] 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.

[33] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume 1. Springer-Verlag, Berlin/New York, 1992.

[34] 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.

[35] José Meseguer. A Logical Theory of Concurrent Objects and Its Realization in the Maude Language. Technical Report SRI-CSL-92-08, SRI International, July 1992.

[36] Brad A. Myers. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. In *UIST–Fourth Annual Symposium on User Interface Software Technology*, pages 211–220, 1991.

[37] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-Oriented Software Development. *Communications of the ACM*, 35(9):160–165, September 1992.

[38] 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.

[39] A.B. Potengy, C.J.P. Lucena, and D.D. Cowan. A Programming Approach for Parallel rendering Applications. Technical report, Monografias em Ciência da Computação, PUC-Rio, April 1993.

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

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

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

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