# The Semantics of Abstract Data Views: A Design Concept to Support Reuse-in-the-Large

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

**Abstract**

In this paper we present a preliminary formal semantics of a new design concept to support reuse-in-the-large called Abstract Data Views (ADVs). This approach toward reuse is described informally followed by the presentation of a formal basis for its semantics. The semantics of the relationship of ADVs and ADTs as well as the semantics of a specification constructor for simple composition are presented. Issues on reusability, the main concern of the presented design concept, and the role of instantiation and naming requirements in this context are discussed. We also indicate how the dynamic features of ADVs and consistency requirements can be represented through a temporal logic formalism. Finally, a proof outline of a modularization theorem for reuse of ADTs interpreted through ADVs is presented.

**Resumo**

Neste artigo nós apresentamos a semântica formal preliminar de um novo conceito de projeto que suporta reuso em ponto grande chamado Visões Abstratas de Dados ("ADVs"). Esta abordagem para o reuso é descrita informalmente seguida de uma apresentação de uma base formal para a sua semântica. A semântica do relacionamento de ADVs e ADTs bem como a semântica de um construtor de especificações para a composição simples são apresentadas. Questões sobre reutilização, a principal preocupaçao do presente conceito de projeto, e o papel da instanciação e dos requisitos para associaçao de nomes são discutidas neste contexto. Nós também indicamos como as características dinâmicas de ADVs e os requisitos de consistência podem ser representados atravës de um formalismo em lögica temporal. Finalmente, o esboço da prova de um teorema de modularizaçao para o reuso de ADTs interpretados através de ADVs é apresentado.

*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 (the Brazilian Research Council). 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

# 1 Motivation

The intent of this extended abstract is to describe the semantics of a new design concept that supports reuse-in-the-large. This approach toward reuse is described informally followed by an outline of a formal basis for its semantics. This formal basis uses constructors to produce complex specifications from simpler components.

DeRemer and Kron [17] have coined the terms programming-in-the-small (the coding of individual modules) and programming-in-the-large (programming at the module level), a term that can be used as a synonym for program design. Therefore, by reuse-in-the-large we mean the reuse of program designs involving the description of module specifications and their interconnections. The design concept to be described in this present abstract is based on the use of Abstract Data Types (ADTs) [32] to model program modules.

Using pairs of ADTs to represent modules and their interfaces in reusable designs provided the original motivation for the concept of Abstract Data Views (ADVs)[11, 12] (pairs of ADTs have also been used in a slightly different sense in [40]). A special ADT, called the ADV, is used as an interface (in a very broad sense) for ADTs in designs and provides a view of an ADT. Specification constructors are used to combine ADVs and ADTs to produce more complex designs, and this process has been validated by proof of concept architectures.

The approach can be seen as a way of providing language support for the specification and abstraction of inter-object behavior [13, 30]. ADVs have been used to support user interfaces for games and a graph editor [10], to interconnect modules in a user interface design system (UIDS) [35], to support concurrency in a cooperative drawing tool, and to implement a ray-tracer in a distributed environment [37]. The VX·REXX [1] system that was built as a research prototype was motivated by the idea of composing applications in the ADV/ADT style. This applied work is closely related to the theoretical activities described in this abstract.

Since we are interested in reuse-in-the-large, we need to combine the specification of modules by guaranteeing syntactic non-interference and semantic context independence [16] among the different module specifications. Combination of specifications is achieved by specification constructors such as simple composition, sets and sequences of component types, and inheritance of specifications [33] that are used for both ADV and ADT specifications. ADTs do not know about their associated ADVs; different ADVs may view the same ADT providing the design satisfies a non-interference proof obligation.

We have recently been investigating the incorporation of the notion of an ADV and its specification constructors into VDM [34]. Although VDM seems adequate to model ADV-based reusable designs, we have been using a complementary visual formalism called ADVcharts [6] based on Statecharts [29] and Objectcharts [9], to provide a formal specification or problem statement for the design. Work is in progress on relating these two linguistic representations to allow proof of the correctness of a textual design based on a visual specification. ADVs are also being used to support the formulation of a Object-Oriented design method; a preliminary version has been compared to similar efforts reported in the literature such as [4, 39].

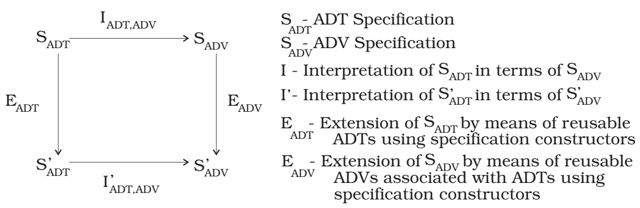


Figure 1: The Modularization Theorem for Reuse of ADTs interpreted through ADVs

This abstract contains a description of a preliminary theory of ADVs to support the work described in the previous paragraphs. We provide the semantics of ADVs and their associated specification constructors and give a proof outline of the central notion of reuse supported by our design concept. We have phrased our central property for reuse-in-the-large in a similar style to the one adopted in [25]: *an ADT specification* *extended by other reusable ADT specifications through the use of a given set of specification constructors can be interpreted (or viewed) as an equivalent ADV specification provided that the original ADT can be interpreted (or viewed) by an associated ADV that is extended by the application of the same given set of specification constructors applied to the ADVs associated with the reused ADTs.* In other words, we need to show that the operators $I$ and $E$ in Figure 1 commute. Using the terminology adopted in [41], we can say that we have a "modularization theorem" for the reuse-in-the-large of ADTs interpreted as ADVs.

In a recently published survey [14] a number of formalisms are discussed in the context of reuse. The formalisms address the compositional and the transformational approaches to reuse. The first supports the development of reusable components [5, 3, 27, 26, 31, 42, 43] and the second supports the reuse of parts of a development process [7, 18, 15, 38]. Our work clearly falls into the first category.

Since the notion of pairs of ADTs (one viewing the other) is central to our work, the logical formalism called "interpretation between theories" [19] appears as the most adequate choice. Nevertheless, to make this first version of the theory of ADVs more readable and intuitive, we have resorted to the simplified style of presenting semantics used by Hoare [32] in his classical seminal paper. For that purpose, we have established the analogy of an ADV as an "abstract program" and its corresponding ADT as a "data representation". In addition, we use states, pre- and post-conditions and invariants in the VDM style [34]. In [32], knowledge about the semantics of Algol was assumed in order to provide the semantics through "textual substitution". In this abstract, we assume knowledge about the semantics of ADVs, which have been formalized in [44] and [28].

# 2 An Informal Overview of the Abstract Data View Concept

The Abstract Data View (ADV) supports a design approach that divides the specification of software modules into two distinct types of components: *Abstract Data Types (ADTs), and Abstract Data Views (ADVs).* An ADV is also an ADT and provides an interface between two ADTs or between an ADT and another medium such as a user or a network. This interface component is called an Abstract Data View (ADV) since it provides a restricted view of an ADT. ADVs were originally conceived as a method of separating the specification of the user interface (UI) components from the non-user-interface (NUI) components of an interactive system and were intended to promote design reuse. Section 3 illustrates the use of ADVs through a number of examples. In the ADV concept, ADTs are completely independent of the interface and have their requirements for data translated by one or more ADVs. In the context of user interfaces this implies that ADTs do not have direct access to input or output.

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

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

The relation between an ADV and an ADT is not symmetric, there is no variable *owner* in the ADT. Since an ADT has no knowledge of input or output, it does not know about its corresponding ADVs. In a design using the ADV concept, a consistency property needs to be satisfied, since several ADV instances

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

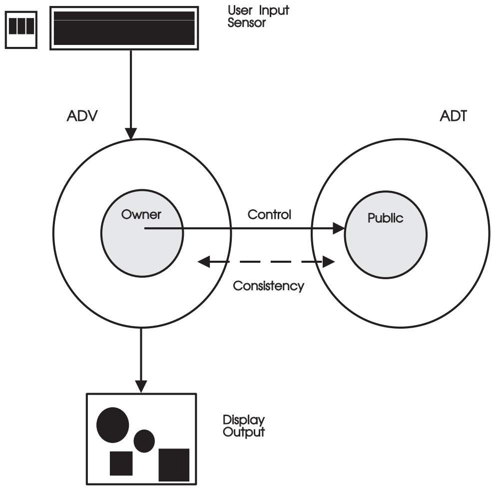


Figure 2: The ADV Model

could be associated with the same ADT instance in order to provide different views or a different control functionality for the same ADT.

The ADV design concept promotes reuse of interface specifications because it allows complex interface components to be built from simpler interface components. Since ADVs represent ADTs, reuse of ADTs is a consequence of this fact. To build complex interface components we use specification constructors based on composition with locality, sets and sequences of components, and inheritance of specifications [33]. In this abstract, we show the formalization of the composition constructor.

# 3 An Overview of ADVs as a General Design Concept

The Abstract Data View (ADV) concept supports the specification of interfaces between media and between Abstract Data Types (ADTs). In this context, ADVs can be viewed as domain transformers and domain constrainers [13].

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

The concept of ADV as a metaphor can also be generalized beyond the user interface; ADVs can be used to map the functionality required by an ADT into the functionality provided by another ADT. This concept is also illustrated in Figure 3 (iii) where the ADV is used to map some of the functionality needed by an electronic mail system into the functionality provided by a file system. ADVs can also be used to specify the conversion between two ADTs that are different representations of the same abstract notion ("twin ADTs" or domain-oriented virtual machines [40]), and also to define a common interface to make the twin ADTs appear as a single ADT. Such a concept is illustrated in Figure 3 (iv).

Although ADVs can be used as general interfaces and metaphors, the ADV can also have added features that can constrain the domain of an ADT. Figure 4 illustrates two ways that this constraint mechanism

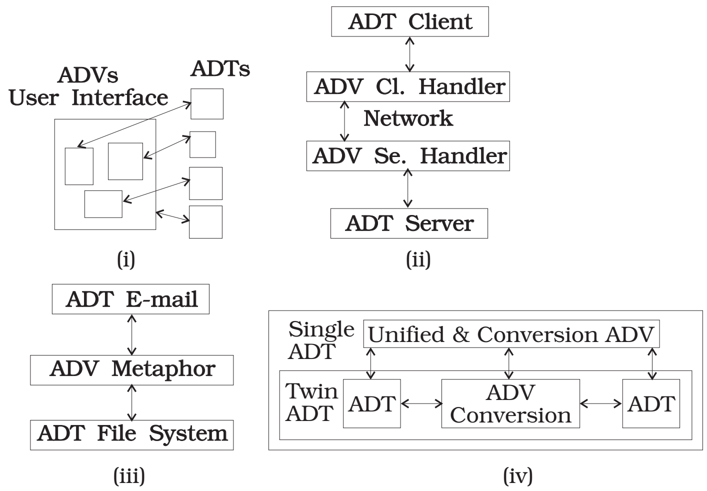


Figure 3: ADVs as Domain Transformers

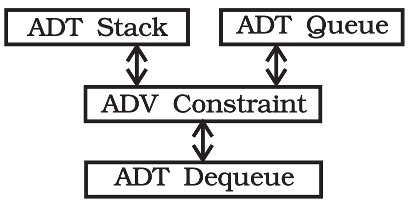


Figure 4: ADVs as Domain Constraints

$ADV\_Type$ = ADV [is ADV] [for ADT]

  declaration $x_i : T_i, (i = 1, \ldots, I)$

  invariant Inv

  component $ADV\_Type = \overline{ADV}$ [for ADV'] [for ADT']

  .......

  component set of $ADV\_Type_1 = \overline{ADV}$ [for ADV'] [for ADT']

  .......

  component seq of $ADV\_Type_2 = \overline{ADV}$ [for ADV'] [for ADT']

  .......

  function $f_j\ (a_{k,j} : U_{k,j}\ ; k = 1, \ldots, K_j)\ a_{k+1,j} : U_{k+1,j}\ (j = 1, \ldots, J)$

    declaration $y_{l,j} : V_{l,j}, (l = 1, \ldots, L)$

    external $status_{m,j} : z_{m,j}, (m = 1, \ldots, M)$,

      where $status_{m,j} \in \{`wr', `rd'\}$

    pre_condition $pre\_ADV\_f_j$

    post_condition $post\_ADV\_f_j$

  event $e_n\ (b_{o,n} : R_{o,n}\ ; o = 1, \ldots, O_n)\ (n = 1, \ldots, N)$

    declaration $v_{p,n} : S_{p,n}, (p = 1, \ldots, P)$

    external $status_{q,n} : w_{q,n}, (q = 1, \ldots, Q)$,

      where $status_{q,n} \in \{`wr', `rd'\}$

    pre_condition $pre\_ADV\_e_n$

    post_condition $post\_ADV\_e_n$

end ADV

Figure 5: A General Schema for an ADV

might be specified. The ADV can be used to hide some of the functionality of an ADT. For example, the diagram shows the ADV making the dequeue ADT appear as both a stack and a queue. Such an approach provides different views of an ADT and is similar to the concept of views in data base design. The ADV can also contain concurrency constraints and limit access to the ADT dequeue by the ADT stack or the ADT queue.

# 4 The Semantics of ADVs and their Specification Constructors

In order to give the semantics of the specification constructor related to composition (the semantics of the other constructors are given in [2]), we first provide the representation of both ADVs and ADTs and give the interpretation of the relationship between ADTs and ADVs through the variable *owner*. A general schema for an ADV is shown in Figure 5.

In this schema, the declaration "$ADV\_Type$ = ADV [is ADV] [for ADT]" expresses the definition of a type ADV. The symbol "[..]" encloses optional syntactic items, that is, the declarations "is ADV" and "for ADT" are optional. The declaration "is ADV" represents the inheritance relationship, that is, the $ADV\_Type$ is being defined as a specialization of another $ADV\_Type$. The declaration "for ADT" represents the association of an $ADV\_Type$ with an $ADT\_Type$.

An $ADV\_Type$ is composed of a declaration part, invariants, components, functions, and events. The declaration part represents the private variables of the $ADV\_Type$. The invariants describe the constraints on variables that compose the $ADV\_Type$. The components represent the structural composition of the $ADV\_Type$. Finally, functions and events describe the behavior of the $ADV\_Type$.

The components of an $ADV\_Type$ can be a simple type (represented by "$ADV\_Type = \overline{ADV}$ [is ADV'][for ADT']"), a set type (represented by "set of $ADV\_Type_1 = \overline{ADV}$ [is ADV'][for ADT']"), or a sequence type (represented by "seq of $ADV\_Type_2 = \overline{ADV}$ [is ADV'][for ADT']"). The notation $\overline{ADV}$ is used to express a name of a particular component of the $ADV\_Type$ that is being defined. The statement "set of $ADV\_Type_1 = \overline{ADV}$ [is ADV'][for ADT']" defines a component with type "set of $ADV\_Type_1$" and "$ADV\_Type_1$" is defined by the expression following the equal sign, that is "$\overline{ADV}$ [is ADV'][for ADT']". This same interpretation is used for the statement "seq of $ADV\_Type_2 = \overline{ADV}$ [is ADV'][for ADT']", where instead of a set we are defining a component with type "seq of $ADV\_Type_2$ ".

Each function $f_j$ defined in an $ADV\_Type$ has parameters $a_{k,j}$ of type $U_{k,j}$ $(k = 1, ..., K_j)$ and returns as result $a_{k+1,j}$ of type $U_{k+1,j}$. For each function $f_j$, we define pre- and post-conditions represented by $pre\_ADV\_f_j$ and $post\_ADV\_f_j$, respectively. In each function, we might define some local variables using the declaration field inside the function, and might also define the external variables that are being affected (status = "wr") or consulted (status = "rd") by the function using the external mechanism inside the function.

Each event $e_n$ defined in an $ADV\_Type$ has parameters $b_{o,n}$ of type $R_{o,n}$ $(o = 1, ..., O_n)$. An event is described in a manner similar to a function, that is, it has a declaration part, an external part, and pre- and post-conditions. The declaration and external parts are also optional.

A general schema for an ADT is presented in Figure 6. The symbols and constructs used in this schema are the same as the ones used in the ADV representation.

$$
\begin{array}{lll}
\underline{ADT\_Type = ADT} & & \\
& \underline{\text{declaration}} & \overline{x}_i : \overline{T}_i,\ (i = 1, \ldots, \overline{I}) \\
& \underline{\text{invariant}} & \overline{Inv} \\
& \underline{\text{component}} & ADT\_Type = \overline{ADT} \\
& & \ldots\ldots \\
& \underline{\text{function}} & \overline{f}_j\ (\overline{a}_{k,j} : \overline{U}_{k,j}\ ;\ k = 1, \ldots, \overline{K}_j)\ \overline{a}_{k+1,j} : \overline{U}_{k+1,j}\ (j = 1, \ldots, \overline{J}) \\
& & \underline{\text{declaration}}\ \overline{y}_{l,j} : \overline{V}_{l,j},\ (l = 1, \ldots, \overline{L}) \\
& & \underline{\text{external}}\ status_{m,j} : \overline{z}_{m,j},\ (m = 1, \ldots, \overline{M}), \\
& & \qquad \text{where } \underline{status_{m,j}} \in \{`wr', `rd'\} \\
& & \underline{\text{pre\_condition}}\ \overline{pre\_ADT\_f_j} \\
& & \underline{\text{post\_condition}}\ \overline{post\_ADT\_f_j} \\
\underline{\text{end}}\ ADT & &
\end{array}
$$

Figure 6: A General Schema for an ADT

Using these ADV and ADT representations, we describe the interpretation of the association of an $ADV\_Type$ with an $ADT\_Type$. The association of an ADV with an ADT is represented by the variable *owner*. We assume that the variable *owner* is of type $ADT\_Type$ and is part of the declaration of an $ADV\_Type$. This assumption is illustrated by the representation in Figure 7.

Note that, in general, the pre- and post-conditions of each event of the abstract data view $ADV$, namely, $pre\_ADV\_e_n$ and $post\_ADV\_e_n$ for an arbitrary $n$, may refer to functions of the abstract data type *owner*. The pre- and post-conditions of each function of the abstract data view $ADV$, namely, $pre\_ADV\_f_j$ and $post\_ADV\_f_j$ for an arbitrary $j$, may also refer to functions of the abstract data type *owner*. However, functions in the ADT may not reference the functions or events in the ADV, since there is not a symmetric representation of the variable *owner* in the ADT.

The interpretation of the abstract data view $ADV$ which has an associated variable *owner* is based on the fact that the behavior of $ADV$ is extended with the behavior of its associated *owner*. In other words, the functionality of the $ADT$ is included in the functionality of the $ADV$. This connection allows the effect

$ADV\_Type$ : $ADV$ for ADT
    declaration $x_i : T_i, (i = 1, \ldots, I)$
        $owner$ : $\underline{ADT\_Type}$
    invariant Inv
    component .......
    function $f_j\ (a_{k,j} : U_{k,j}\ ; k = 1, \ldots, K_j)\ a_{k+1,j} : U_{k+1,j}\ (j = 1, \ldots, J)$
        declaration $y_{l,j} : V_{l,j}, (l = 1, \ldots, L)$
        external $status_{m,j} : z_{m,j}, (m = 1, \ldots, M)$,
            where $status_{m,j} \in \{\text{'}wr'\text{, '}rd'\}$
        pre_condition $pre\_ADV\_f_j$
        post_condition $post\_ADV\_f_j$
    event $e_n\ (b_{o,n} : R_{o,n}\ ; o = 1, \ldots, O_n)\ (n = 1, \ldots, N)$
        declaration $v_{p,n} : S_{p,n}, (p = 1, \ldots, P)$
        external $status_{q,n} : w_{q,n}, (q = 1, \ldots, Q)$,
            where $status_{q,n} \in \{\text{'}wr'\text{, '}rd'\}$
        pre_condition $pre\_ADV\_e_n$
        post_condition $post\_ADV\_e_n$
end ADV

Figure 7: A General Schema Showing Inclusion of an ADT in an ADV

of each function $\overline{f}_j$ defined in the ADT represented by the variable *owner* to be quoted in the pre- and post-conditions of the events and functions of the ADV. In this way, the behavior of the $\overline{f}_j$s are embedded as extra effects of the pre- and post-conditions of the events and functions of the ADV. The interpretation is based on the notion of textual substitution [32] and is outlined in the following steps:

1. The declaration *owner* : *ADV_Type* is regarded as equivalent to the body of the declaration of the *ADT_Type* : *ADT* after the occurrence of its variables, invariant, functions and pre- and post-conditions have been prefixed by '*owner*'. Thus, if the $\overline{ADT\_Type}$ is represented as in the Figure 6, *owner* : $\overline{ADT\_Type}$ is equivalent to

    declaration $owner\_\overline{x}_i : \overline{T}_i, (i = 1, \ldots, \overline{I})$
    invariant $\overline{owner\_Inv}$
    function $owner\_\overline{f}_j\ (\overline{a}_{k,j} : \overline{U}_{k,j}\ ; k = 1, \ldots, \overline{K_j})\ \overline{a}_{k+1,j} : \overline{U}_{k+1,j}\ (j = 1, \ldots, \overline{J})$
        declaration $owner\_\overline{y}_{l,j} : \overline{V}_{l,j}, (l = 1, \ldots, \overline{L})$
        external $owner\_status_{m,j} : owner\_\overline{z}_{m,j}, (m = 1, \ldots, \overline{M})$,
            where $\overline{owner\_status_{m,j}} \in \{\text{'}wr'\text{, '}rd'\}$
        pre_condition $\overline{pre\_owner\_f_j}$
        post_condition $\overline{post\_owner\_f_j}$

2. Furthermore, all variables, and pre- and post-conditions of the functions of the variable *owner* in an *ADV* are prefixed by "*owner*". For instance, the post-condition $\overline{post\_f_j}$ of the function $\overline{f}_j$ (that belongs to the *ADT owner*) is renamed $\overline{post\_owner\_f_j}$.

3. If a pre- or a post-condition of a function $\overline{f}_j$ of the variable *owner*, $\overline{pre\_owner\_f_j}$ or $\overline{post\_owner\_f_j}$, is used in the pre-condition of a particular function $f_j$ or event $e_n$ of the *ADV*, then:

    (a) The formal parameters of $\overline{pre\_owner\_f_j}$ and $\overline{post\_owner\_f_j}$ are replaced by the actual parameters, resulting in $\overline{Act\_pre\_owner\_f_j}$ and $\overline{Act\_post\_owner\_f_j}$.

(b) These pre- and post-conditions of a funtion $\overline{f}_j$ after the substitutions described previously are replaced by their associated logical expressions according to their definitions in the variable *owner* and the resulting expressions are denoted by $\overline{Exp\_pre\_owner\_f_j}$ and $\overline{Exp\_post\_owner\_f_j}$, respectively. The steps (a) and (b) can also be applied in the case where the functions of the $ADV$ are used in the pre- and post-conditions of the events or other functions of the $ADV$ if we use their associated logical expressions that are stated in their definitions in the same $ADV$.

(c) The resulting pre-conditions, after the previous steps have been completed for both the function $f_j$ and the event $e_n$ denoted, respectively, by $New\_pre\_ADV\_f_j$ and $New\_pre\_ADV\_e_n$, are related to the pre- and post-conditions of the function $\overline{f}_j$ of the $ADT$ *owner* in the following way:

$$
\begin{aligned}
&New\_pre\_f_j \rightarrow \overline{Exp\_post\_owner\_f_j}\\
&New\_pre\_f_j \rightarrow \overline{Exp\_pre\_owner\_f_j}\\
&New\_pre\_e_n \rightarrow \overline{Exp\_post\_owner\_f_j}\\
&New\_pre\_e_n \rightarrow \overline{Exp\_pre\_owner\_f_j}
\end{aligned}
$$

4. If a pre- or a post-condition of a function $\overline{f}_j$ of the variable *owner*, $\overline{pre\_owner\_f_j}$ or $\overline{post\_owner\_f_j}$, is used in the post-condition of a particular function $f_j$ or event $e_n$ of the $ADV$, then:

   (a) The steps (2a) and (2b) described previously are completed.

   (b) The resulting post-conditions, after the above steps have been completed for both the function $f_j$ and the event $e_n$ denoted, respectively, by the expressions $New\_post\_ADV\_f_j$ and $New\_post\_ADV\_e_n$, are related to the pre- and post-conditions of the function $\overline{f}_j$ of the $ADT$ represented by the variable *owner* in the following way:

$$
\begin{aligned}
&New\_post\_ADV\_f_j \rightarrow \overline{Exp\_pre\_owner\_f_j}\\
&New\_post\_ADV\_f_j \rightarrow \overline{Exp\_post\_owner\_f_j}\\
&New\_post\_ADV\_e_n \rightarrow \overline{Exp\_pre\_owner\_f_j}\\
&New\_post\_ADV\_e_n \rightarrow \overline{Exp\_post\_owner\_f_j}
\end{aligned}
$$

Let us now present the interpretation of the composition of two ADVs. Assume that an abstract data view $\overline{ADV_Y}$ is a component of an abstract data view denoted by $ADV_X$. This is represented in Figures 8 and 9.

The interpretation of the composite abstract data view $ADV_X$ that has $\overline{ADV_Y}$ as a component is based on the fact that the behavior of $ADV_X$ is extended with the behavior of $\overline{ADV_Y}$. This idea of inclusion of behavior is essentially the same idea used to model the relationship between an ADV and an ADT. The interpretation is also based on textual substitution and is outlined in the following steps:

1. The declaration <u>*component*</u> $ADV\_Type\_Y \ : \ \overline{ADV_Y}$ is regarded as equivalent to the body of the declaration of the $ADV\_Type\_Y \ : \ ADV_Y$ after the occurrence of its variables, invariant, functions and pre- and post-conditions have been prefixed by '$\overline{ADV_Y}$'. Thus, if we assume the schema of Figure 9 for the $ADV\_Type\_Y \ : \ ADV_Y$, then <u>*component*</u> $ADV\_Type\_Y \ : \ \overline{ADV_Y}$ is equivalent to

| | |
|---|---|
| <u>declaration</u> | $\overline{ADV_Y\_x_i} : \overline{T}_i, \ (i = 1, \ldots, \overline{m})$ |
| <u>invariant</u> | $\overline{ADV_Y\_Inv}$ |
| <u>function</u> | $\overline{ADV_Y\_f_j} \ (\overline{a}_{k,j} : \overline{U}_{k,j} \ ; k = 1, \ldots, \overline{K}_j) \ \overline{a}_{k+1,j} : \overline{U}_{k+1,j} \ (j = 1, \ldots, \overline{J})$ |
| | <u>declaration</u> $\overline{ADV_Y\_y_{l,j}} : \overline{V}_{l,j}, \ (l = 1, \ldots, \overline{L})$ |
| | <u>external</u> $\overline{ADV_Y\_status_{m,j}} : \overline{ADV_Y\_z_{m,j}}, \ (m = 1, \ldots, \overline{M})$, |
| | where $\overline{ADV_Y\_status_{m,j}} \in \{‘wr’, ‘rd’\}$ |
| | <u>pre_condition</u> $\overline{pre\_ADV_Y\_f_j}$ |

$ADV\_Type\_X \; = \; ADV_X$

$\underline{\text{declaration}}$ $x_i : T_i,\ (i = 1, \ldots, I)$

$\underline{\text{invariant}}$ Inv

$\underline{\text{component}}$ $ADV\_Type\_Y : \overline{ADV_Y}$

$\underline{\text{function}}$ $f_j\ (a_{k,j} : U_{k,j}\ ;\ k = 1, \ldots, K_j)\ a_{k+1,j} : U_{k+1,j}\ (j = 1, \ldots, J)$

$\underline{\text{declaration}}$ $y_{l,j} : V_{l,j},\ (l = 1, \ldots, L)$

$\underline{\text{external}}$ $status_{m,j} : z_{m,j},\ (m = 1, \ldots, M)$,

where $status_{m,j} \in \{`wr', `rd'\}$

$\underline{\text{pre\_condition}}$ $pre\_ADV_X\_f_j$

$\underline{\text{post\_condition}}$ $post\_ADV_X\_f_j$

$\underline{\text{event}}$ $e_n\ (b_{o,n} : R_{o,n}\ ;\ o = 1, \ldots, O_n)\ (n = 1, \ldots, N)$

$\underline{\text{declaration}}$ $v_{p,n} : S_{p,n},\ (p = 1, \ldots, P)$

$\underline{\text{external}}$ $status_{q,n} : w_{q,n},\ (q = 1, \ldots, Q)$,

where $status_{q,n} \in \{`wr', `rd'\}$

$\underline{\text{pre\_condition}}$ $pre\_ADV_X\_e_n$

$\underline{\text{post\_condition}}$ $post\_ADV_X\_e_n$

$\underline{\text{end}}$ $ADV_X$

Figure 8: An ADV of Type X with an ADV of Type Y as a Component

$ADV\_Type\_Y \; = \; ADV_Y$

$\underline{\text{declaration}}$ $\overline{x}_i : \overline{T}_i,\ (i = 1, \ldots, \overline{I})$

$\underline{\text{invariant}}$ $\overline{Inv}$

$\underline{\text{function}}$ $\overline{f}_j\ (\overline{a}_{k,j} : \overline{U}_{k,j}\ ;\ k = 1, \ldots, \overline{K_j})\ \overline{a}_{k+1,j} : \overline{U}_{k+1,j}\ (j = 1, \ldots, \overline{J})$

$\underline{\text{declaration}}$ $\overline{y}_{l,j} : \overline{V}_{l,j},\ (l = 1, \ldots, \overline{L})$

$\underline{\text{external}}$ $status_{m,j} : \overline{z}_{m,j},\ (m = 1, \ldots, \overline{M})$,

where $\overline{status_{m,j}} \in \{`wr', `rd'\}$

$\underline{\text{pre\_condition}}$ $\overline{pre\_ADV_Y\_f_j}$

$\underline{\text{post\_condition}}$ $\overline{post\_ADV_Y\_f_j}$

$\underline{\text{event}}$ $\overline{e}_n\ (\overline{b}_{o,n} : R_{o,n}\ ;\ o = 1, \ldots, \overline{O_n})\ (n = 1, \ldots, N)$

$\underline{\text{declaration}}$ $\overline{v}_{p,n} : \overline{S}_{p,n},\ (p = 1, \ldots, \overline{P})$

$\underline{\text{external}}$ $status_{q,n} : \overline{w}_{q,n},\ (q = 1, \ldots, \overline{Q})$,

where $\overline{status_{q,n}} \in \{`wr', `rd'\}$

$\underline{\text{pre\_condition}}$ $\overline{pre\_ADV_Y\_e_n}$

$\underline{\text{post\_condition}}$ $\overline{post\_ADV_Y\_e_n}$

$\underline{\text{end}}$ $ADV_Y$

Figure 9: An ADV of Type Y

| | |
|---|---|
| | <u>post_condition</u> $\overline{post\_ADV_Y\_f_j}$ |
| <u>event</u> | $\overline{ADV_Y\_e_n}$ $(\overline{b_{o,n}} : R_{o,n}\ ; o = 1, \ldots, \overline{O_n})$ $(n = 1, \ldots, \overline{N})$ |
| | <u>declaration</u> $\overline{ADV_Y\_v_{p,n}} : \overline{S_{p,n}}$, $(p = 1, \ldots, \overline{P})$ |
| | <u>external</u> $\overline{ADV_Y\_status_{q,n}} : \overline{ADV_Y\_w_{q,n}}$, $(q = 1, \ldots, \overline{Q})$, |
| | where $\overline{ADV_Y\_status_{q,n}} \in \{`wr', `rd'\}$ |
| | <u>pre_condition</u> $\overline{pre\_ADV_Y\_e_n}$ |
| | <u>post_condition</u> $\overline{post\_ADV_Y\_e_n}$ |

2. Furthermore, all variables, and pre- and post-conditions of the functions of $\overline{ADV_Y}$ referred to in $ADV_X$ are prefixed by "$\overline{ADV_Y}$" and the state space of $ADV_X$ is extended with the variables declared in $\overline{ADV_Y}$ prefixed in the same way. For instance, if a variable $x$ of the $ADV_Y$ occurs in $ADV_X$, it is renamed "$\overline{ADV_Y\_x}$".

3. If a pre- or a post-condition of a function $\overline{f_j}$ of the $\overline{ADV_Y}$, namely $\overline{pre\_ADV_Y\_f_j}$ and $\overline{post\_ADV_Y\_f_j}$, are used in the pre-condition of a particular function $f_j$ or event $e_n$ of the $ADV_X$, then:

   (a) The formal parameters of $\overline{pre\_ADV_Y\_f_j}$ and $\overline{post\_ADV_Y\_f_j}$ are replaced by the actual parameters, resulting in $\overline{Act\_pre\_ADV_Y\_f_j}$ and $\overline{Act\_post\_ADV_Y\_f_j}$, respectively.

   (b) These pre- and post-conditions of $\overline{f_j}$ after the substitution previously described are replaced by their associated logical expressions according to their definitions in the $\overline{ADV_Y}$ and the resulting expressions are denoted by $\overline{Exp\_pre\_ADV_Y\_f_j}$ and $\overline{Exp\_post\_ADV_Y\_f_j}$. The steps (a) and (b) described previously are also applied in the case where the functions of the $ADV_X$ are used in the pre- and post-conditions of the events or other functions of $ADV_X$.

   (c) The resulting pre-conditions after the previous steps have been completed for both the function $f_j$ and the event $e_n$ which are denoted, respectively, by the expressions $New\_pre\_ADV_X\_f_j$ and $New\_pre\_ADV_X\_e_n$, are related to the pre- and post-conditions of the function $\overline{f_j}$ of the $\overline{ADV_Y}$ in the following way:

   $New\_pre\_ADV_X\_f_j \rightarrow \overline{Exp\_pre\_ADV_Y\_f_j}$
   $New\_pre\_ADV_X\_f_j \rightarrow \overline{Exp\_post\_ADV_Y\_f_j}$
   $New\_pre\_ADV_X\_e_n \rightarrow \overline{Exp\_pre\_ADV_Y\_f_j}$
   $New\_pre\_ADV_X\_e_n \rightarrow \overline{Exp\_post\_ADV_Y\_f_j}$

4. If a pre- or a post-condition of a function $\overline{f_j}$ of the $\overline{ADV_Y}$, namely $\overline{pre\_ADV_Y\_f_j}$ and $\overline{post\_ADV_Y\_f_j}$, are used in the post-condition of a particular function $f_j$ or event $e_n$ of the $ADV_X$, then:

   (a) The steps (2a) and (2b) described previously are completed.

   (b) The resulting post-conditions, after the above steps have been completed for both the function $f_j$ and the event $e_n$ which are denoted, respectively, by the expressions $New\_post\_ADV_X\_f_j$ and $New\_post\_ADV_X\_e_n$, are related to the pre- and post-conditions of the function $\overline{f_j}$ of the $\overline{ADV_Y}$ in the following way:

   $New\_post\_ADV_X\_f_j \rightarrow \overline{Exp\_pre\_ADV_Y\_f_j}$
   $New\_post\_ADV_X\_f_j \rightarrow \overline{Exp\_post\_ADV_Y\_f_j}$ $New\_post\_ADV_X\_e_n \rightarrow \overline{Exp\_pre\_ADV_Y\_f_j}$
   $New\_post\_ADV_X\_e_n \rightarrow \overline{Exp\_post\_ADV_Y\_f_j}$

Note that when an $ADT$ is related to an $ADV$ through the variable *owner*, the renaming process (textual substitution by prefixing) guarantees the non-interference of the particular element *owner* of type $ADT$ related to the $ADV$ with the other elements of the same type $ADT$ (non-interference of multiple views).

Similarly, when an element of type $ADV_Y$ is used as a component of another type $ADV_X$, the renaming process also guarantees the non-interference of this particular element of type $ADV_Y$ with the other elements of type $ADV_Y$ (visibility).

# 5 Issues on Reusability: Instantiation and Parametrization (Naming)

A significant barrier to the reuse of both designs and implementations of software components is the fact that they acquire knowledge about their surrounding environment in such a way that clean separation of components is difficult. Such depth of specialized knowledge seems counter not only to reuse but to good engineering practice in general. For example, 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. In other words a component should not be aware of the syntactic or semantic structure of a component from which it acquires services. It follows that a disciplined approach to naming among components is a prerequisite to reuse of component specifications or implementations. Particularly, the software components need to acquire data from both the operating environment and their associated components and to maintain a high-degree of reusability.

Since the operating environment is common to all components, its syntactic and semantic structure can be completely hidden by the use of a consistent level of abstraction. For example, if a user interface object wants to determine if an input event is relevant, it can call the operating environment with a function and an argument which contains the object's own name. The operating environment can return a boolean value. Since all the user interface objects can use this same abstraction, their specifications or implementations should be reusable in the context of this operating environment. Of course if other operating environments provide similar abstractions then the user interface object is reusable across all such operating environments.

Consider components $A$ and $B$ where $A$ is a client of $B$. The interfaces specifying the service request from $A$ and the service provision from $B$ may not match because we have reused these components. In order to make the two interfaces match we need a mapping "component" that will transform the syntactic and semantic structure of the service request onto the syntactic and semantic structure of the service provision. This special mapping "component" will not necessarily be reusable, but will act as a transformational glue to link the two reusable components $A$ and $B$ together. When we consider specifications this special "component" is an $ADV$, whereas at the implementation level this component could be viewed as the realization of the $ADV$ as a powerful linking mechanism.

We now examine the composite components $A$ and $B$ and their constituents. The mechanism described in the previous paragraph might be used to link constituents into their associated composite components, but in many cases, simpler mechanisms might be sufficient. If $A$ and $B$ only need to ask their components to perform certain actions, the constituent is completely reusable since $A$ or $B$ is responsible for mapping the request into the syntactic or semantic structure of the constituent at the time of request. If a component $A$ or $B$ needs data from its parent then this data could be supplied through an initialization parameter that is in the specification or implementation. The interpretation of this parameter is that the parent provides the data when the component is instantiated. Thus, in this case, the parent is completely responsible for transforming the request into the syntactic or semantic structure of the component and so the component is completely reusable.

It should be clear that the design has to proceed in a methodical way in order to satisfy all the naming constraints mentioned earlier. Let us consider first the naming in composite objects. When a component object that is part of a composite object, is specified then we have to specify a method for the component to acquire the name of the composite object and any related parameters. A component can be a single object or a set or sequence of objects. The component object is one of several components that will be grouped together to form a composite object. Hence a composite can be pictured as a container or parent for each of its composite objects. Since the composite object creates the instances of the components this name and the name of any related parameters can be provided when the component is instantiated. This parameter passing mechanism could have the form:

$$component.Init(parent(p_1,\ p_2,\ \ldots,\ p_n)),$$

where parent is the name of the composite object and the parameters $p_i$ are the names and types of the variables in the composite object. Components and composite objects can be *ADT*s or *ADV*s. All objects are components of some composite object; components which do not belong to a composite object are a component of the operating environment or "system".

Let us consider now the naming in *ADV*s. There are two classes of ownership implied in the *ADV/ADT* relationship. An *ADV* can own an entire *ADT* and represent its data structure on the screen, or *ADV* can own an *ADT* because it needs access to some or all of its operations. When an *ADV* is associated with an *ADT* then we have to provide a method for the *ADV* to acquire the name of its owner of associated *ADT* composite object. An *ADV* can be a single object or a set of sequence of objects. Since the *ADT* will be formed before its associated *ADV* the name of the owner *ADT* can be provided through a parameter when the *ADV* is instantiated. Since the *ADT* creates the instances of its *ADV*s the name of the owner and the name of any related parameters can be provided when the component is instantiated. This parameter passing mechanism could have the form:

$$ADV.Init(\ owner(o_1,\ o_2,\ \ldots,\ o_n),\ parent(p_1,\ p_2,\ \ldots,\ p_n)),$$

where owner is the name of the corresponding *ADT* and the parameters $o_i$ are the names and types of the operations in the owner *ADT*. Since an *ADV* can also be a component of a composite object then the *ADV* initialization could also have the parent parameters.

Note that when the specification for an object is written the interaction with other objects must be carefully specified in order to limit naming issues that could reduce reusability.

# 6 Consistency Properties of ADVs and ADTs

The semantics of the relationship between and ADT and an ADV and the semantics of the presented specification constructors were given in terms of a textual substitution process. In fact, this semantics only cope with the static aspects of the specification, describing the input-output behavior of function and events related to particular ADTs or ADVs. In order to present the semantics of the dynamic aspects of the specification of ADVs we are going to use a temporal logic formalism [36].

We will assume, as usual, that an assertional language is used to describe the properties of states. We will refer to a formula in this language as a state formula, or simply as an assertion. A temporal formula is constructed out of state formulas to which the boolean operators $\neg$ and $\vee$ (the other boolean operators can be defined from these; in fact other sets of basic operators can be used as, for instance, $\neg$ and $\rightarrow$) and the following basic temporal operators can be applied:

$\bigcirc$ - Next $\quad\mathcal{U}$ - Until $\quad\ominus$ - Previous $\quad\mathcal{S}$ - Since

Other temporal operators can be defined using the basic set of temporal operators as follows: $\Diamond\phi = \top\mathcal{U}\phi$ (eventually, where $\top$ means true), $\Box\phi = \neg\Diamond\neg\phi$ (henceforth), $\phi\mathcal{W}\psi = \Diamond\phi \vee (\phi\mathcal{U}\psi)$ (Waiting-for, Unless or Weak Until), $\diamondminus\,\phi = \top\mathcal{S}\phi$ (Sometimes in the past), $\boxminus\,\phi = \neg\diamondminus\,\neg\phi$ (Always in the past), $\phi\mathcal{B}\psi = \boxminus\,\phi \vee (\phi\mathcal{S}\psi)$ (Back-to or Weak Since), $\overline{\ominus}\phi = \neg\ominus\neg\phi$ (Weak Previous).

The dynamic aspects of the specification of ADVs can be stated by means of a classification of temporal logic properties [8]. According to this classification we have safety, guarantee, obligation, response, persistence and reactivity formulas. We can also treat liveness formulas. Safety properties can be stated by safety formulas, i.e. formulas of the form $\Box\phi$, where $\phi$ is a past formula (i.e., a formula that does not contain the operators $\bigcirc$ and $\mathcal{U}$). These formulas state that all positions in a computation satisfy $\phi$ or, equivalently, all proper prefixes of a computation end-satisfy $\phi$. Safety formulas can be used, for instance, to express invariance of some state property over all computations.

We consider a simplified (operating) environment with a set of $ADV$s, denoted by $SET_{ADV}$, and a set of $ADT$s, denoted by $SET_{ADT}$, and some implicit functionality. This functionality of the (operating) environment, as noticed previously, is common to all components of the system. For simplicity, we shall omit details of this functionality and concentrate only on the vertical and horizontal consistency requirements for the system. We have required that these $ADV$s and $ADT$s are instantiated in accordance with the naming requirements described earlier.

The ADV concept allows the association of several $ADV$ instances with a single $ADT$ instance, where each $ADV$ instance can provide a different view of the $ADT$ instance or a different control functionality. An $ADV$ instance is associated with one and only one $ADT$ instance at a time (an $ADV$ instance has only one *owner* at a time). On the other hand, an $ADT$ instance can be associated with several $ADV$ instances. Thus, an $ADT$ instance can be viewed in different ways, but the different views represented by the multiple $ADV$ instances must be consistent with the $ADT$ instance they represent.

Vertical consistency involves the relationship between and $ADV$ and an $ADT$ through the variable *owner*. The vertical consistency property states that for all $ADV$s of the environment, there is an $ADT$ of the environment such that throughout system execution the results of the function applications of the $ADV.owner$ are equal to the results of the function applications of the $ADT$. This property can be expressed by the formula

$$\begin{array}{l}\forall\, adv \in SET_{ADV}\ .\ \exists\, adt \in SET_{ADT}\ .\\ \Box\, \forall\, j\ \forall\, x_i, x_i'\ (i = 1, \ldots, k)\ .\ [adv.owner.f_j'(x_1', \ldots, x_k')\ =\ adt.f_j(x_i, \ldots, x_k)]\end{array}$$

where $adv.owner.f_j'(x_1', \ldots, x_k')$ are the functions that belong to $adv.owner$ of the $adv$ and $adt.f_j(x_i, \ldots, x_k)$ are the functions that belong to the public interface of the *adt*.

Horizontal consistency is related to the fact that an $ADV$ instance is associated with one and only one $ADT$ instance at a time, i.e., an $ADV$ instance has only one owner at a time. This property can be expressed by the formula

$$\begin{array}{l}\forall\, adv_1, adv_2 \in SET_{ADV}\ .\ \forall\, adt \in SET_{ADT}\ .\\ \Box\, \forall\, j\ \forall\, x_i, x_i'(i = 1, \ldots, k)\ .\ [adv_1.owner.f_j'(x_1', \ldots, x_k')\ =\ adt.f_j(x_1, \ldots, x_k) \wedge\\ adv_2.owner.f_j''(x_1'', \ldots, x_k'')\ =\ adt.f_j(x_1, \ldots, x_k)\\ \rightarrow\ adv_1.owner.f_j'(x_1', \ldots, x_k')\ =\ adv_2.owner.f_j''(x_1'', \ldots, x_k'')]\end{array}$$

Note that this property requires that for each particular time throughout system execution, there is only one *adv* associated with an *adt*, because if the content of both the records $adv_1.owner$ and $adv_2.owner$ are equal to the public interface of the *adt*, then they must be equal.

# 7 Proof Outline for the Modularization Theorem for Reuse

The proof of the Modularization Theorem for reuse of ADTs interpreted through ADVs is illustrated in Figure 1 and outlined in the following steps. This proof is done by cases, one for each specification construct that was presented, i.e., composition, inheritance, sets and sequences. However, as the specification constructs for sets and sequences can be seen as multiple and ordered multiple composition, and, furthermore, as the specification construct for inheritance can be interpreted as composition through inheritance, all the specification constructors are composition-based [2] and so we will present only the proof for the construct for composition. The proof for the other cases are straightforward extensions of this proof.

The following proof is based on the interpretation of the relationship of an ADV and its variable *owner* which is of *ADT_Type*, and the specification construct for composition. Suppose that we have two ADVs

$ADV_X$ and $ADV_Y$ and that their associated ADTs are denoted, respectively, by $ADT_X$ and $ADT_Y$. This proof essentially consists of five steps:

1. The first step is related to the interpretation $I_{ADT,ADV}$. In this case, $S_{ADV}$ can be specified in terms of $S_{ADT}$ through the connection between the ADV and the ADT by means of the variable *owner*. Particularly, the $ADV_X$ is associated with $ADT_X$ through a variable *owner* (which is of type $ADT_X\_Type$);

2. The second step is related to the extension $E_{ADV}$. In this case $S_{ADV}$ is extended to an $ADV_Y$ associated with an $ADT_Y$ using the specification constructors in order to obtain $S'_{ADV}$. In this case we use the fact that $ADV_X$ can be a component of $ADV_Y$;

3. The third step is related to the extension $E_{ADT}$. In this case $S_{ADT}$ can be extended by composition to the $ADT_Y$ associated with $ADV_Y$ to get a new $S'_{ADT}$. Thus we can compose $ADT_X$ and $ADT_Y$ and get $S'_{ADT}$, which is the composite $ADT_Y$;

4. The fourth step is related to the interpretation $I'_{ADT,ADV}$. In this case the extended $S'_{ADT}$ can be used in the specification of $S'_{ADV}$. We now associate the composite $ADT_Y$ with an $ADV_Y$ to get another specification $S'_{ADV}$, assuming that $ADV_X$ is a component of $ADV_Y$;

5. In the last step we must prove that the steps (1-2) and (3-4) lead to a $S'_{ADV}$ with the same functionality.

In what follows we use the following notation for conciseness. The variables and functions of the $ADT_X$ will be denoted, respectively, by $x$ and $f$, while the variables and functions of the $ADT_Y$ will be denoted by $\overline{x}$ and $\overline{f}$, respectively. On the other hand, the variables, functions and events of the $ADV_X$ will be denoted by $x'$, $f'$ and $e'$, respectively, while the variables, functions and events of $ADV_Y$ will be denoted by $\overline{x'}$, $\overline{f'}$ and $\overline{e'}$, respectively.

In order to compare the functionality of the specifications $S'_{ADV}$ obtained as a result of the steps 1-2 and 3-4 above, we are going to consider the individual functionality of the two ADTs and the two ADVs:

1. As the result of steps 1-2 we have the following function and event pre- and post-conditions:

$$
\begin{array}{l}
cond\_ADV_X\_ADT_X\_f_j(\ldots cond\_ADV_X\_ADT_X\_f_s\ldots) \\
\overline{cond\_ADT_Y\_f}_j(\ldots \overline{cond\_ADT_Y\_f}_{\overline{s}}\ldots cond\_ADT_X\_f_s\ldots) \\
cond\_ADV_X\_(f'_j/e'_n)(\ldots cond\_ADT_X\_f_s \ldots cond\_ADV_X\_f'_{s'} \ldots) \\
\overline{cond\_ADV_Y\_(f'_j/e'_n)}(\ldots \overline{cond\_ADT_Y\_f}_{\overline{s}}(\ldots cond\_ADT_X\_f_s\ldots \overline{cond\_ADT_Y\_f}_{\overline{s}})\ldots \\
\qquad\qquad \ldots cond\_ADV_X\_f'_{s'}(\ldots cond\_ADT_X\_f_s\ldots cond\_ADV_X\_f'_{s'})\ldots \\
\qquad\qquad \ldots cond\_ADV_Y\_f_{\overline{s'}}\ldots)
\end{array}
$$

2. The following function and event pre- and post-conditions that result from steps 3-4 are (respectively) equivalent to the ones obtained as the result of steps 1-2:

$$
\begin{array}{l}
cond\_ADT_Y\_ADT_X\_f_j(\ldots cond\_ADT_Y\_ADT_X\_f_s\ldots) \\
\overline{cond\_ADT_Y\_f}_j(\ldots cond\_ADT_X\_f_s\ldots \overline{cond\_ADT_Y\_f}_{\overline{s}}\ldots) \\
cond\_ADV_X\_(f'_j/e'_n)(\ldots cond\_ADT_X\_f_s\ldots cond\_ADV_X\_f'_{s'}\ldots) \\
\overline{cond\_ADV_Y\_(f'_j/e'_n)}(\ldots \overline{cond\_ADT_Y\_f}_{\overline{s}}(\ldots cond\_ADT_X\_f_s\ldots \overline{cond\_ADT_Y\_f}_{\overline{s}}\ldots)\ldots \\
\qquad\qquad \ldots cond\_ADV_X\_f'_{s'}(\ldots cond\_ADT_X\_f_s\ldots cond\_ADV_X\_f'_{s'}\ldots) \\
\qquad\qquad \ldots cond\_ADV_Y\_f_{\overline{s'}}\ldots)
\end{array}
$$

This implies that the $S'_{ADV}$ obtained as the result of the steps 1-2 and 3-4, respectively, have the same functionality. See [2] for the details.

We should note here that a full proof of the modularization theorem can be given when we formalize the component-of relationship through theories and interpretation between theories. In this context we can define logical interpretations and extensions related with ADVs and ADTs and use interpretation between theories in order to prove results similar to this theorem. The formal semantics is based on theory presentations in a logic developed to support the required object-oriented specification mechanisms and will allow us to formalize the specification constructors described here [20, 21, 24, 22, 23].

# 8 Conclusions and Future Work

This paper has described the preliminary semantics for the ADV design concept which supports reuse-in-the-large. The semantics for ADVs were given in terms of textual substitution in the same style used by Hoare in [32]. Lack of space prevented us from presenting the semantics of all the specification constructors used in the theory but the reader can infer the process (the semantics of all the constructors are presented in [2]). Because of the reduced amount of detail provided in this paper it was only possible to present a proof outline for the central theorem of the theory called the modularization theorem for reuse of ADTs interpreted through ADVs. We believe the outline is sufficient to convince the reader that the detailed proof provided in [2] can easily be constructed because of the style of the semantic definitions that have been used.

The notation in which the semantics is expressed in this preliminary version of the theory was chosen for its readability thus providing a clear understanding of the intuitive aspects of the concepts. In a next step we will go further by capturing the semantics in logic by using interpretation between theories. This kind of formalization which was also used in [41] 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 indicate in [2] how this style of formalization follows almost directly from the one presented in this paper.

# References


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

[2] 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.

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

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

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

[6] L. M. F. Carneiro-Coffin, D. D. Cowan, and C. J. P. Lucena. ADVcharts: a Visual Formalism for Highly Interactive Systems. In M. D. Harrison and C. W. Johnson, editors, *Software Engineering in Human-Computer Interaction.* Cambridge University Press, 1995. To appear.

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

[8] E. Chang, Z. Manna, and A. Pnueli. The Safety-Progress Classification. Technical Report 92-1408, Department of Computer Science, Stanford University, Stanford, California, USA, February 1992.

[9] 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.

[10] 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.

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

[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 Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, November 1993.

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

[15] J. Cramer et al. Towards a Formally Based Component Description Language - a Foundation for Reuse. *Structured Programming*, 12(2):91–110, April 1991.

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

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

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

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

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

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

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

[23] 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.

[24] 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.

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

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

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

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

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

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

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

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

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

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

[35] 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.

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

[37] 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.

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

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

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

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

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

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

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