DESIGN VIEWS FOR SYNTHESIS

Providing Both Uniform Data Integration and Diverse Data Customization

Elke A. Rundensteiner
University of Michigan, Ann Arbor
Dept. of Elect. Eng. and Computer Science
Ann Arbor, MI 48109-2122
e-mail: rundenst@eecs.umich.edu
phone: 313-936-2971

Abstract

Synthesis is a complex task spanning many levels of abstractions and information domains. Hence, CAD systems utilize a global design database to achieve the much needed integration of this diverse design information into one central data model. Such a central database represents a serious bottleneck for the CAD system. First, it prevents the extensibility of the CAD system over time, since a change of the global data model requires an (often prohibitively expensive) modification of all current design tools using the database. Second, it forces all design tools to work on the same (comprehensive and hence extremely complex) data model. In this paper, we introduce a solution to this problem. We propose to utilize the object-oriented view methodology, called MultiView, for specifying customized tool interfaces (design views) on the CAD database. A design view contains a subset of relevant information from the global database organized in a fashion most suitable to the needs of a particular design tool. MultiView automatically maintains the mapping between the global data model and local design views, thus freeing individual design tools from this burden. Our approach thus results in a flexible CAD environment that assures the consistent integration of design data from different tools, while providing each tool with a customized view of the integrated data. This paper gives numerous examples that demonstrate MultiView and its advantages for typical tasks in high-level synthesis.

1 INTRODUCTION

As pointed out in [13], up to 40 percent of a group's engineering resources are expended on tool integration, which is often more than the capital spent on acquiring the tools themselves. For this reason, the CAD Framework Initiative [13, 9] focuses on issues such as standard data exchange formats, reference data models, and standard tool programming interfaces. While the development of standards in data exchange is an important task, universally accepted standards are the exception rather than the rule [11]. Also, they are always being made obsolete by the fast pace of change in IC technology and the type of design tools.

For this reason, the approach of standard file formats (and the corresponding pairs of file translators) is viable at best for exchanging data between different stand-alone CAD systems (from different vendors). Within a CAD system, on the other hand, it is preferable to have design tools operate on one centralized database. Such a unified CAD database is essential for insuring the proper integration of design information generated and consumed by various design tools into one consistent design.

Note that each synthesis task or tool needs a subset of the information maintained by the global database, usually organized to suit the tool's particular requirements. For this reason, we need to be able to support customized design information. Hence, a central database represents a bottleneck of a CAD system, requiring all design tools to work on the same global data model. The global data model is comprehensive and generic: the former implies that individual tools have to wade through huge amounts of irrelevant data to find data of interest to them; and the latter implies that the data is generally not organized in a format appropriate for accomplishing the particular task of a design tool.

In short, we are faced with the following conflicting goals:

- the *integration* and *unification* of the diverse design data into one global database to explicitly capture all relationships between the diverse pieces of the design, and
- the *customization* of the global design data into special-purpose data structures to store the data needed for a particular design task in a format most suitable for the task at hand.

Existing CAD databases do not address this problem of mismatch between the global database and the local data needs of individual tools. They thus put the burden of mapping between these two models onto individual tools. The result of this generally is that tools will load the data from the central database into a design file; and then they will generate a local data structure suitable for their particular design task in their workspace. This approach is undesirable for many apparent reasons: not only does it place an unnecessary burden on the design tool builders, but it also results in a performance penalty and possibly a loss of information due to the translation between incompatible models.

In this paper, we propose a solution to this problem. Namely, we propose to extend the database with the functionality necessary for automatically supporting this mapping between the global model and the local tool interfaces. More precisely, we introduce a mechanism that supports the principled generation of customized tool interfaces (design views) to the database. This provides the CAD framework developer with a methodology for supporting multiple views for a wide range of design tasks. A design view generally contains a subset of relevant information from the global database reorganized in a fashion most suitable to

the needs of particular users. The proposed mechanism, the object-oriented view methodology called MultiView [26, 25], establishes a consistent mapping between the database and the tool interfaces, such that updates through the tool interface are consistently reflected in the underlying database. This strategy promises to eliminate the bottleneck, allowing each tool to work on their customized view of the data while the database assures the consistent integration of the inputs from different tools into one consistent global model. The support for the generation of tool interfaces will dramatically improve the development time of tools, since a major portion of tool development is generally spent on designing and implementing tool interfaces.

The issue of how the shared database can be extended as new design tools, and with them new data requirements, are being added to the system is another open problem. If a change of the global object model is required due to the introduction of a new tool, then all current tools using the database may have to be revised. This is clearly extremely costly and therefore prevents the *extensibility* of the system over time. Until this issue of system extensibility has been successfully addressed, the CAD industry will be hesitant to utilize database technology in their CAD systems. In this paper, we provide a solution to this problem by shielding tools from changes in the global data model. In fact, we will demonstrate that the proposed view-based approach also solves this problem of extensibility. In short, as long as the view on which a particular tool operates is maintained, the tool is not affected by any changes to the global database.

The process of specifying views requires a language for virtual class derivation and for view schema definition. These languages are used to declaratively specify what design information should be in a particular view and how the information should be organized. MultiView provides such languages for deriving local views of global design information. More precisely, MultiView breaks view specification into three tasks: (1) customization of virtual classes, (2) integration of virtual classes into *one* consistent global schema and (3) the specification of arbitrarily complex view schemata on this global schema. MultiView's division of view specification into a number of well-defined tasks, some of which have been successfully automated, makes it a powerful tool for supporting the specification of views by non-database experts while enforcing view consistency. This paper briefly introduces MultiView and the languages for view specification. Emphasis is placed on their utility in the context of high-level synthesis. In particular, we present numerous examples of design views constructed for typical tasks in high-level synthesis.

The rest of the paper is organized as follows. In Section 2, we introduce related work in the fields of CAD and databases. Section 3 describes view specification in general and MultiView in particular. This includes a brief presentation of the object algebra, the class integration algorithm, and the view specification language. Section 4 presents examples that demonstrate the application of MultiView to high-level synthesis. An evaluation of the view-based approach is presented in Section 5, while Section 6 provides conclusions.

2 PREVIOUS WORK

2.1 Design databases and CAD Frameworks

Over the last years, many different data models and design databases have been introduced to support the process of design [10, 11, 14, 15, 2, 3, 6, 21]. Figures 1.a to 1.c demonstrate how this research has gone through several phases, namely, from the storage of design data in flat files (Figure 1.a), to the usage of the traditional database technology, in particular, the relational model (Figure 1.b), up to the adoption of the advanced database systems

such as the object-oriented ones for the maintenance of the design information (Figure 1.c). In this paper, we are not concerned with developing yet another design data model or a design database system. On the contrary, we propose a general mechanism for support of design views that could be added to any of the existing design databases. As discussed in the previous section, view support would enhance the power of any of the existing CAD database systems.

In general, design databases have become a fundamental component of a typical CAD framework [23, 22]. However, all of the existing CAD databases leave the burden of mapping between the tool's local data structures and the database's global data model with the individual design tools. In this paper, on the other hand, we address this problem by extending the database with the necessary functionality required for supporting this mapping. In other words, we will move the responsibility of mapping between the global database and the local tool models to the database (Figure 1.d). We thus extend the notion of CAD frameworks to include a view-based database as central component. To our knowledge, this is the first time that database view mechanisms have been explored to be applied to CAD.

2.2 Design views in CAD

There is some work in the CAD literature that uses the term 'views', however, the associated meaning and with it the usage of the view concept is different than proposed in this paper [20, 19]. Sometimes the different information domains of an application, such as the behavioral domain and the structural domain in behavioral synthesis, are referred to as *views* of the design [19]. Note, however, that these two information domains are two different parts of the global schema, each captured by different data structures. The behavioral-graph and the structural-graph thus are trivial views in MultiView; they both correspond to simple subsets of the complete information model. Furthermore, note that the structural view (domain) of the design is generated from the behavioral view (domain) using design tools, i.e., there are actual design decisions involved in generating one from the other. Hence, given a behavioral view of a design, the database would and should not be powerful enough to construct a structural view – rather the generation of these different (information models) views is what the "process of design" is all about.

In OCT [16], the term view is used to refer to different facets of a frame, each storing independent pieces of information about the design like the physical characteristics, the behavior, etc. For this reason, this does not correspond to the concept of design views (of viewing the same design information in polymorphic ways) as presented in this paper. To summarize, we propose a general methodology for declaratively specifying new design views (and to get the associated mappings to the global database for free), whereas other work has built fixed tool interfaces for a particular design task in a rather ad-hoc manner. Other systems can therefore not quickly construct new design views nor easily modify existing ones.

2.3 On Object-Oriented View Mechanisms

View mechanisms for object-oriented databases have been identified as one of the few open problems in object-oriented database research in [4]. Recently, there have been several proposals for defining views for OODBs [18, 28, 1, 25]. Most of them use the query language defined for their respective object model as view specification language, namely, for deriving a virtual class. They generally do not discuss the integration of derived classes into the global schema nor do they generate complete view schemata. Instead, the derived classes

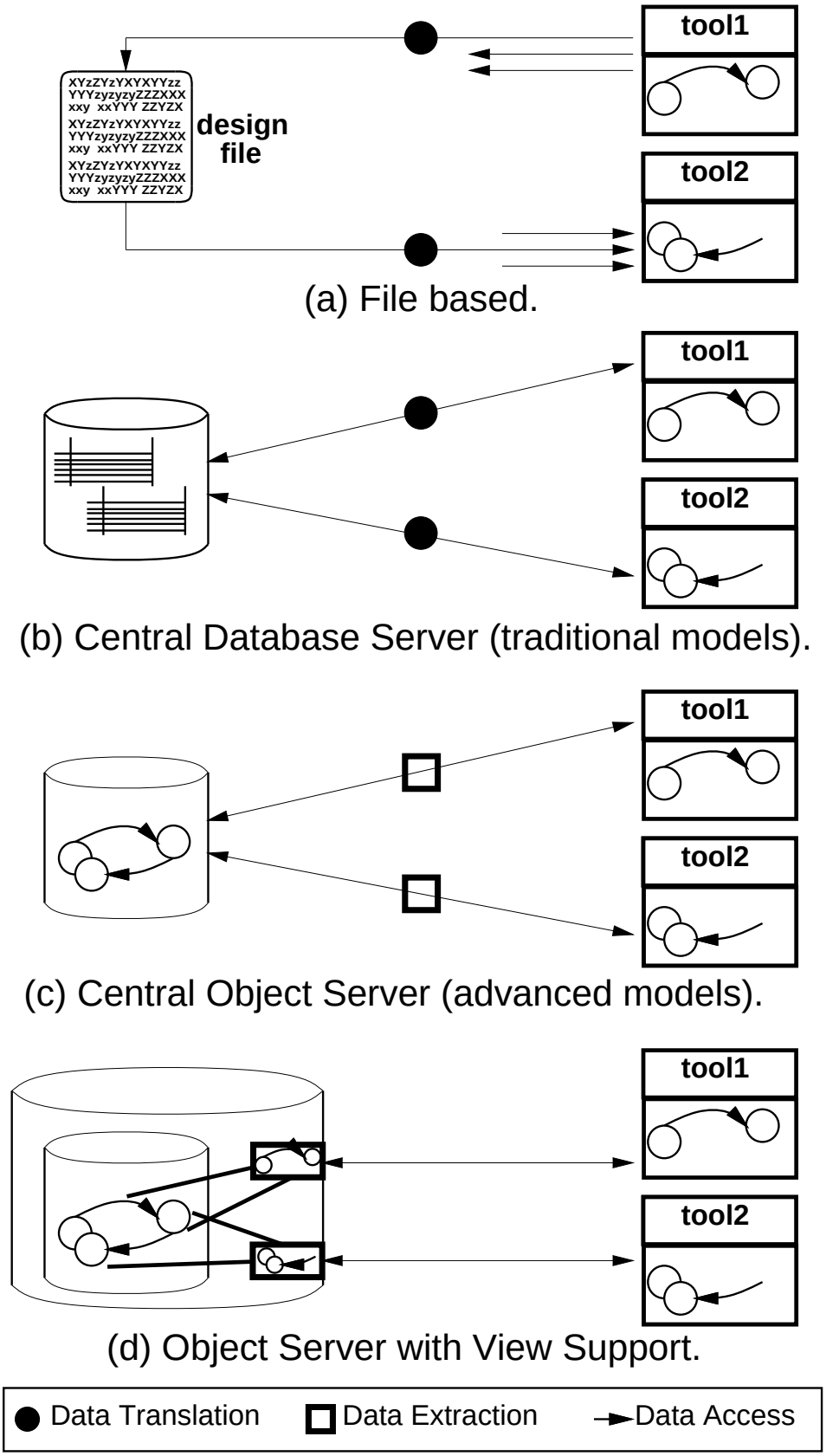


Figure 1: Approaches towards Tool Integration.

are treated as 'stand-alone' objects [18] or they are attached directly as subclasses of the schema root. MultiView is one of the only view methodologies that generates complete view schemata. Secondly, algorithms for automating the more tedious parts of view specification have been developed for MultiView. And, lastly, MultiView takes care of enforcing the consistency of the view schema. For these reasons, we have chosen MultiView as the most suitable candidate for this work.

2.4 Relational Views

While object-oriented views are a new database technology, the concept of views for relational databases has been studied extensively by the database community. The question thus arises why relational views have not been proposed as tool integration scheme at the time when relational database technology was suggested for implementing CAD databases. The remainder of this section provides an answer to this question.

One reason is the limit of the *semantic modeling* power of the relational model itself. CAD databases based on the relational model capture the design information in a rigid tabular format, which, being machine-oriented, does not correspond to a direct model of the real-world design information (Figure 1.b). This gap between the rigid database model and the structurally much richer design information of course also applies to relational views; i.e., there is a semantic gap between the relational database views and the information models required by design tools. Therefore, CAD tools would not directly operate on relational views anyways; and hence the purpose of relational views would be void.

The second reason is the *view update ambiguity* problem of relational databases. Since updates on relational views can generally not be translated into (unambiguous) updates on the base schema, most relational databases do not permit update on views. With updating and refining of design information being an essential ingredient of design databases, relational views have consequently been found to be of limited use for applications like CAD. Views in OODBs are more likely to play an important role for defining customized interfaces for CAD applications, since updates can be handled better due to the following: (1) object identity; the concept of maintaining the unique identity of an object even if its external characteristics are modified and/or hidden (in a view), and (2) abstract data types; the ability to associate type-specific (update) operations with the encapsulated object [26].

2.5 Summary

In summary, there has been no consistent and systematic scheme for generating and maintaining customized design views on a central database. This paper is the first that addresses this problem and that suggests the use of object-oriented views for the construction of these customized tool interfaces in CAD. In fact, we postulate that view support will represent an important component of future CAD framework technology.

3 DESIGN VIEW SPECIFICATION

In this section, we will introduce the basic issues of view support in object-oriented databases in general, and an introduction to our solution towards view support, called MultiView, in particular. Emphasis of this paper is on the potential exploitation of this database view technology for addressing the tool integration problem in CAD systems. Therefore, this

section gives a general introduction to of MultiView, while a more detailed description can be found elsewhere [25, 26].

For clarity sake, we distinguish between three groups of humans involved in a CAD system: the design tool builder, the CAD framework maintainer, and the CAD framework builder. The CAD framework builder is in charge of constructing the software components that compose a CAD framework. This includes the development of a user interface, a design process manager, and a design database [23]. We propose that a view support system should be a component of such a CAD framework. Once a CAD framework has been selected as basis for a particular CAD system, then the design tool builder will utilize the functionalities of the framework to develop particular design tools that will 'live' within the CAD framework. We now introduce a middle man, called the CAD framework maintainer, who will use the view support system to construct design views for particular design tools (or related tool sets).

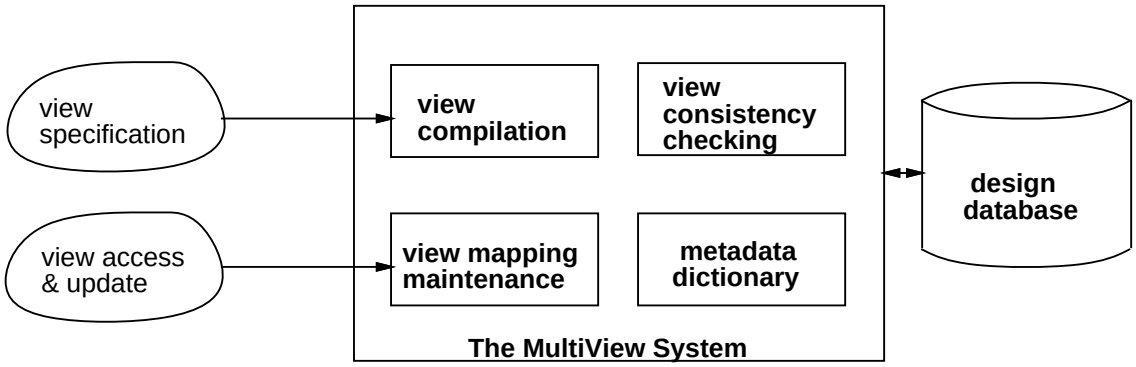


Figure 2: Architecture of the View Support System.

An overview of the architecture of MultiView is given in Figure 2. MultiView consists of:

1. a view specification language,
2. a view compilation module,
3. a view mapping maintenance module,
4. a view consistency checking module, and
5. meta data.

The CAD framework maintainer will enter a view specification into the view support system to specify a new design view. View compilation then processes the view specification. For this purpose, it will use the metadata information (data dictionary) to check for the definition of class names and to establish the necessary information about the new view classes. This phase also involves automatic view generation, where view generation algorithms are run to complete the view description, if necessary [26]. For instance, if the view definer determines that the modify-component-type function must be deleted from the floorplanning view, then the system will automatically generate a new Component class with the modified functions. And, the system will also determine how this new class can be integrated into the global class hierarchy. Once a view has been established, then the designer

and/or design tool will operate on the view in the same manner as on the database, i.e., through a programming interface with access and update functions (Figures 2 left bottom).

The CAD framework maintainer uses the view specification language, while the rest of the system will be hidden from him or her. For this reason, we will concentrate in the remainder of this section on this view specification process. Information on the other parts of MultiView can be found in [26].

3.1 View Specification in MultiView

MultiView is a methodology for supporting multiple view schemata in OODBs. MultiView breaks view specification into three independent tasks:

1. the generation of customized classes,
2. the integration of derived classes into *one* consistent global schema graph, and
3. the specification of arbitrarily complex view schemata composed of both base and virtual classes on top of the augmented global schema.

The separation of the view design process into these well-defined tasks has several advantages. First, it simplifies view specification, since each of the tasks can be solved independently from the others. Second, it increases the level of support by allowing for the automation of some of the tasks. For the first task, MultiView provides an object algebra (see Section 3.3). For the second task, an algorithm has been developed that assures the automatic and consistent integration of all virtual classes into the global database schema. For the third task, MultiView provides a view definition language and an associated view completion algorithm. A detailed presentation of these languages and algorithms can be found in [25], while a brief introduction to each will be given in the remainder of this section. The example given next presents an overview of the different steps of view specification in MultiView.

Example 1. *In Figure 3 (and in the remainder of the paper), we depict base and virtual classes by circles and dotted circles, respectively. Given the global schema GS in Figure 3.a, the view definer specifies the virtual classes* ***ALU*** *and* ***LogicUnits*** *using object-oriented queries (Figure 3.b). The integration of these classes into GS is given in Figure 3.c. View schema definition now proceeds by selecting a subset of classes from the augmented GS (Figure 3.d). Lastly, the chosen view classes are interconnected into one view schema (Figure 3.e) by the view completion algorithm.*

3.2 Basic Terminology On Object-Oriented Views

Since MultiView is based on a fairly typical object model, the reader is referred to a standard book on object-oriented databases for a thorough coverage of this subject [8]. Only the terminology needed for the remainder of the paper is introduced below. A **class** $C_i \in$ C has a unique class name, a type description and a set membership. The type associated with a class, **type**(C), consists of a number of property functions, **properties**(C). A property function could be a value from a simple enumeration type, an object instance from some class, an arbitrarily complex function, or an object method. A **class** is also a container for a set of objects instances that belong to the class, denoted by **extent**(C). For two classes

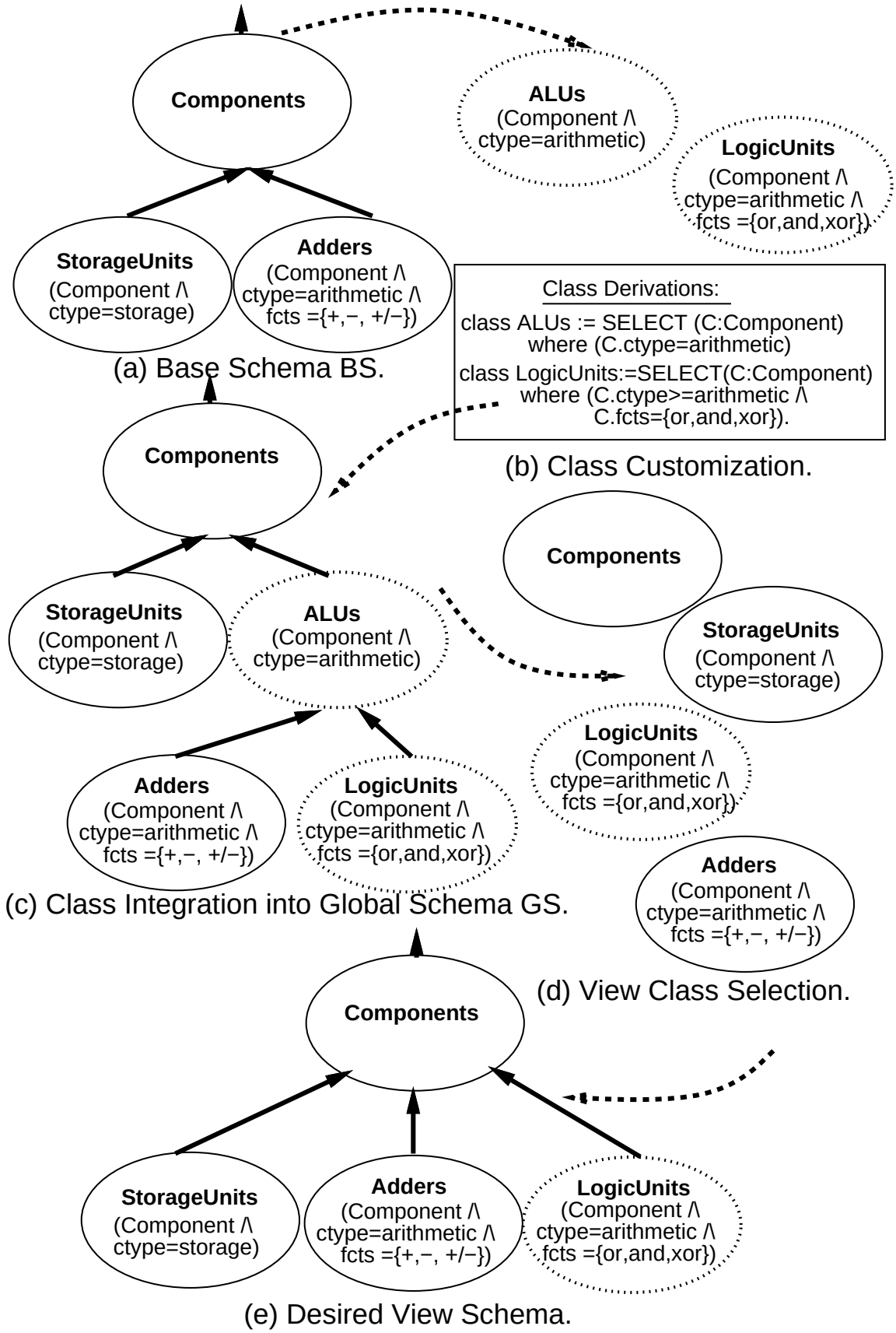


Figure 3: The MultiView Approach: From Base over Global to View Schemata.

C1 and C2 $\in$ C, C1 is called a **subset** of C2, denoted by C1 $\subseteq$ C2, if and only if ($\forall$ o $\in$ O) ((o$\in$C1) $\Longrightarrow$ (o$\in$C2)). For two classes C1 and C2 $\in$ C, C1 is called a **subtype** of C2, denoted by C1 $\preceq$ C2, if and only if (**properties**(C1) $\supseteq$ **properties**(C2)) and ($\forall$ p $\in$ **properties**($C2$)) (**domain**$_p$($C1$) $\subseteq$ **domain**$_p$($C2$)). For two classes C1 and C2 $\in$ C, C1 is called a **subclass** of C2, denoted by C1 *is-a* C2, if and only if (C1 $\preceq$ C2) and (C1 $\subseteq$ C2).

Definition 1. *An **object schema** is a directed acyclic graph S=(V,E), where V is a finite set of vertices and E is a finite set of directed edges. Each element in V corresponds to a class C_i, while E corresponds to a binary relation on $V \times V$. Each directed edge e from C_1 to C_2, denoted by e = <C_1,C_2>, represents the direct is-a relationship (C_1 is-a C_2).*

We refer to the set of *is-a* relationships of a schema as the **generalization hierarchy**. We distinguish between **base** and **virtual** classes. **Base** classes are defined during the initial schema definition and their object instances are explicitly stored as base objects. **Virtual** classes are defined during the lifetime of the database using some object-oriented queries, i.e., their definitions are dynamically added to the existing schema. A virtual class has an associated membership derivation function that will determine its membership based on the state of the database. The content of a virtual class is generally not explicitly stored, but rather computed upon demand.

Definition 2. *The **base schema** (BS) is an object schema S=(V,E), where all classes in V correspond to base classes with stored rather than derived instances. The **global schema** (GS) is an extension of the base schema BS augmented by the collection of all virtual classes defined during the lifetime of the database as well as their is-a relationships. Given a global schema GS=(V,E), then a **view schema** (VS), or short, a **view**, is defined to be a schema VS= (VV,VE) with (1) VS has a unique view identifier < VS >, (2) VV $\subseteq$ V, and (3) VE $\subseteq$ transitive-closure(E).*

We call the classes in a view schema (both the base and virtual ones) *view classes* and the *is-a* relationships *view is-a relationships*. At any given time, there will always be exactly one base and one global schema but an arbitrary number of view schemata.

Example 2. *Figure 3.a shows the base schema BS, Figure 3.c the global schema GS, and Figure 3.e a view schema VS.*

3.3 Task 1: Object Algebra for the Virtual Class Derivation

The first task of view specification in MultiView uses an object algebra [26]. The object algebra provides basic operators that can be used to define new virtual classes based on existing information in the database. These operators can be nested to form arbitrarily complex class derivations. The object algebra consists of six basic operators that are briefly described below.

The **hide** operator modifies the type description of a class by hiding some of its property functions. It has the syntax "<*virtual*-class> = **hide** [<prop-functions>] **from** (<*source*-class>)" with <prop-functions> being one or more property functions defined for <*source*-class>. It removes the property functions listed in the set <prop-functions> from the source class while preserving all others. The set content of the virtual class is equal to the set content of the source class.

The **refine** operator refines the type description of an existing class by adding additional property functions. It has the syntax "<*virtual*-class> = **refine** [<prop-function-defs>] **for** (<*source*-class>)" with <prop-function-def> the definition of a new property function in the form of a new property name and a function body with the latter a legal arithmetic, boolean or set expression. The set content of the virtual class is equal to the set content of the source class.

The **select** operator selects a subset of object instances from a given set of objects. It has the syntax "<*virtual*-class> = **select from** (<*source*-class>) **where** (<predicate>)" with <predicate> being some possibly complex function on the source class and its type description. Its semantics are to return a subset of object instances of the source class based on the evaluation of the associated predicate, namely, all object instances that satisfy the predicate are collected into the virtual class. The type stays the same.

Set operators manipulate both the type description and the set membership of their two source classes. The semantics of the **union** operator are to return a set of object instances composed of the members of either or both of the source classes. The resulting type description is equal to the lowest common supertype of the two sources classes. The **intersect** operator returns a set of object instances that are members of both source classes. Furthermore, the type description of the resulting virtual class is equal to the greatest common subtype of the two sources classes. Lastly, the **difference** operator returns a set of object instances that are members of the first but not of the second source class. The resulting type description is equal to the description of the first source class.

Example 3. *In Figure 4, the is-a relationships between the virtual and the sources classes are indicated by bold arrows. Figure 4.a depicts the query "**BehaviorGraph** = **hide** [SetState, GetState] **from** (**StateGraph**)". Then **extent**(**BehaviorGraph**) = **extent**(**StateGraph**) and **type**(**BehaviorGraph**) = [Domain, NodeOp].*

*In Figure 4.b, the query "**Comps2** = **refine** [Area = Height * Width] **for** (**Comps**)" derives **Comps2**. We have **extent**(**Comps2**) = **extent**(**Comps**). The type of **Comps2** has been extended by the new method Aera, hence **Comps2** $\preceq$ **Comps**. **Comps2** is integrated into GS by placing **Comps2** below **Comps** as direct subclass.*

*In Figure 4.c, the query "**Adders** = **select from** (**Comps**) **where** (Plus in **Comps**.Ops)" derives **Adders** from **Comps**. The **Adders** class consists of all object members of **Comps** that implement the Plus operator, thus **Adders** $\subseteq$ **Comps**. **Type**(**Adders**) = **type**(**Comps**).*

*In Figure 4.d, the query "**GraphConstructs** = **union**(**DataFlow**,**ControlFlow**)" derives **GraphConstructs**. Then extent(GraphConstructs) = extent(DataFlow) $\cup$extent(ControlFlow) = $\{D1, D2, D3, C1, C2\}$.Alsotype(GraphConstructs) = **type**(**DataFlow**) $\sqcap$ **type**(**ControlFlow**) = [Domain]. The is-a relationships are indicated by the edges (**DataFlow** is-a **GraphConstructs**) and (**ControlFlow** is-a **GraphConstructs**).*

*In Figure 4.e, the **intersect** operator is used in the query **FexLayout** = **intersect**(**DataPathUnits**,**RandomLogicUnits**). Then **extent**(**FexLayout**) = **extent**(**DataPathUnits**) $\cap$ **extent**(**RandomLogicUnits**) = $\{O1, O2\}$. And **type**(**FexLayout**) = **type**(**DataPathUnits**) $\sqcup$ **type**(**RandomLogicUnits**)= [CompType, DF-Construct, CF-Construct, get-DF-Graph].*

*In Figure 4.f, the **diff** operator is used in "**AllOtherComps** = **diff**(**Components**,**ALUs**)" to derive **AllOtherComps** from **Components** that are not in **ALUs**. We have **extent**(**AllOtherComps**) = **extent**(**Components**) –*

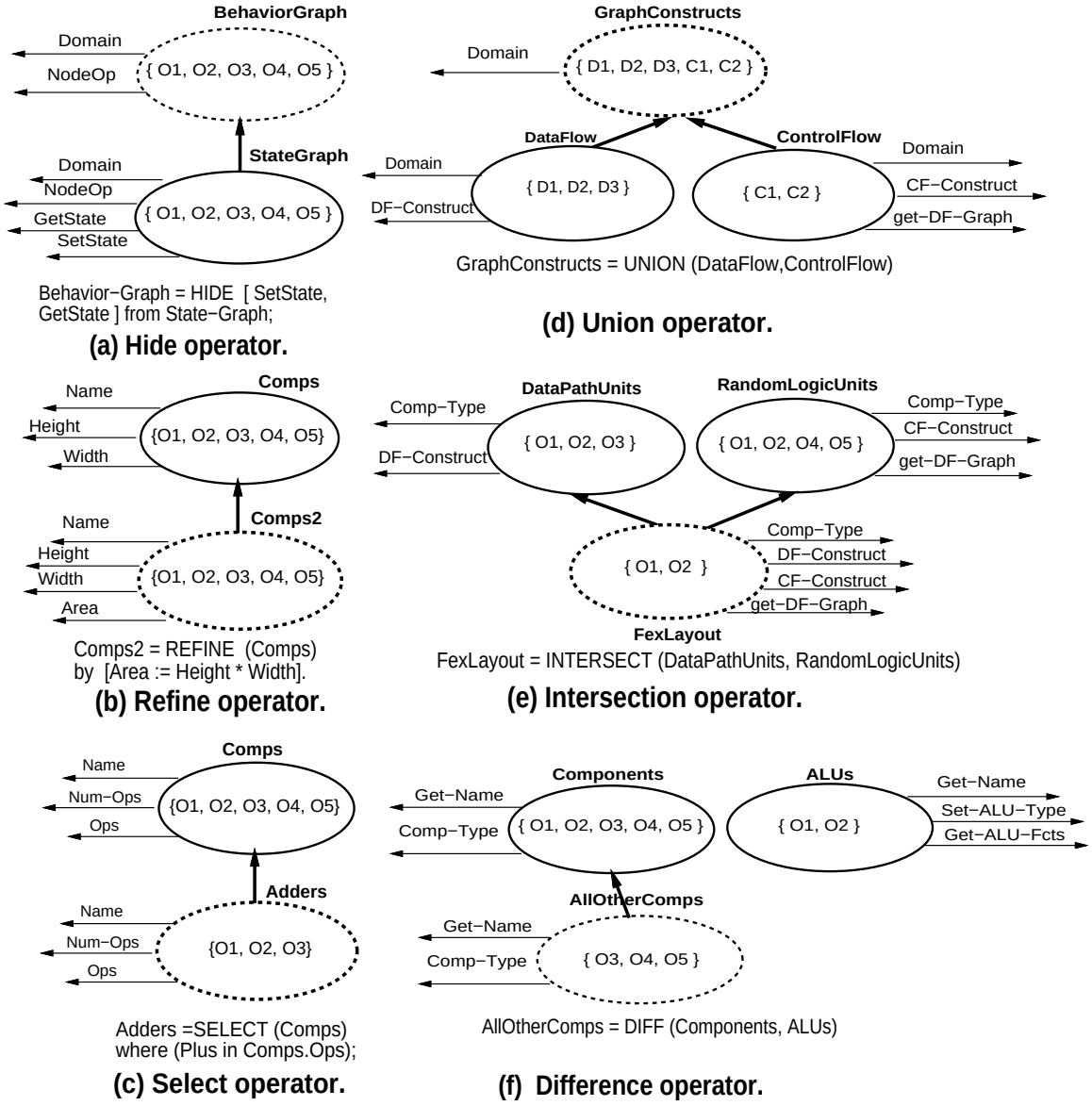


Figure 4: Examples of Class Derivation Using Object Algebra.

extent(ALUs) $= \{O3, O4, O5\}$. *And* ***type(AllOtherComps) = type(Components)*** *= [Get-Name, Comp-Type]. The relationship (****AllOtherComps*** *is-a* ***Components****) has been added to Figure 4.f.*

3.4 Task 2: Class Integration Into The Global Schema

MultiView integrates all virtual classes derived for different views into one global schema in order to explicitly represent the generalization relationships between virtual and base classes. In this section we sketch an overall approach for the class integration problem. A detailed treatment of this topic is beyond the scope of this paper and can be found elsewhere [26].

Class integration is concerned with finding the most 'appropriate' location in the schema graph G for a virtual class VC in terms of property inheritance and subset relationships between classes. For this, the classifier determines the *is-a* relationships between the virtual class VC and all other classes in GS by comparing their type descriptions and their membership predicates[1]. The algorithm for finding the correct position for VC in G=(V,E) can be summarized as follows. First, we find all classes in G that are the direct superclasses of VC defined by direct-parents(VC) $= \{C_i \mid (VC \text{ is-a } C_i) \wedge (\nexists C_j \in V)(j \neq i)((VC \text{ is-a } C_j) \wedge (C_j \text{ is-a } C_i))\}$. Similarly, we find all classes in G that are the direct subclasses of VC defined by direct-children(VC) $= \{C_i \mid (C_i \text{ is-a } VC) \wedge (\nexists C_j \in V)(j \neq i)((C_i \text{ is-a } C_j) \wedge (C_j \text{ is-a } VC))\}$. VC is placed directly below all classes in the direct-parents set and directly above all classes in the direct-children set. Edges connecting classes in the direct-children(VC) set with classes in the direct-parents(VC) set are removed, since these relationships are now represented indirectly via VC. We complete this section by demonstrating the classification process on a simple example.

Example 4. *In Figure 3.b, the virtual class* ***ALU*** *is derived using the query "****ALU = select from (Component) where*** *(ctype=arithmetic))". We can thus deduce the relationships: (****ALU*** $\subseteq$ ***Component****), (****ALU*** $\preceq$ ***Component****), and (****ALU*** *is-a* ***Component****). We therefore insert the edge (****ALU*** *is-a* ***Component****) into GS. Next, we search for the most specialized classes that are still is-a related with the* ***ALU*** *class. We find that the* ***Adders*** *class is both a subset and a subtype of the* ***ALU*** *class; therefore we add the is-a relationship (****Adders*** *is-a* ***ALU****) in form of an edge to the graph.*

3.5 Task 3: View Schema Definition Using the View Definition Language

MultiView divides the third task of view specification into two subtasks:

1. the selection of view classes, and
2. the generation of view relationships between the view classes.

[1] In general, the classification problem is not decidable for OODB models since it may involve the comparison of arbitrary functions and predicates. In the worst case, if some *is-a* relationship is not discovered, then the virtual class is placed higher in the class hierarchy than would theoretically be possible. This would be a correct but not the most informative class arrangement.

For the first subtask, it provides a view definition language that can be utilized by the view definer for the specification of view schemata. For the second subtask, it provides algorithms that will automatically generate a generalization hierarchy from a given set of view classes [26]. This automatic generation of view *is-a* arcs is preferable over their manual entry since it simplifies the task of the view designer and guarantees the consistency of the resulting view schema [26].

The view definition language consists of two groups of operators: the first group initiates or terminates a transaction on a view schema while the second group discussed in the next paragraph modifies a given view schema. The `DEFINE-VIEW` command initializes a new view schema and assigns a unique view identifier to it, while the `MODIFY-VIEW` command prepares an already defined view schema for modification. All operators specified within a view definition transaction, i.e., after a `DEFINE-VIEW` or a `MODIFY-VIEW` command and before the `END-VIEW` command, will modify only the designated view VS. The view definers conclude the view definition phase by issuing the `SAVE-VIEW` command. MultiView then automatically augments the set of classes by the necessary view *is-a* arcs [26].

The second group of commands modifies the view VS by adding or deleting view classes. The "`ADD-CLASS(<`*class*`-name>)`" command adds a class <*class*-name> to VS. The "`ADD-CLASS-DAG(<`*class*`-name>)`" command adds all classes to VS that are classes in the subschema of GS rooted at the class <*class*-name>. Finally, the "`ADD-VIEW-SCHEMA(<`*view*`-name>)`" command adds all classes of the view <*view*-name> to VS. The commands `REMOVE-CLASS, REMOVE-CLASS-DAG,` and `REMOVE-VIEW-SCHEMA` do the same as the just described operators but rather than adding they delete the respective classes.

Example 5. *A **view creation script** for the view VS depicted in Figure 3.e is given below.*

```
DEFINE-VIEW VS
   class ALU = select (C:Component) where (C.ctype=arithmetic);
   class LogicUnits = select (A:ALU)
      where (A.ctype=arithmetic) and (A.fcts={or,and,xor});
   ADD-CLASS (LogicUnits);
   ADD-VIEW-SCHEMA (BS);
   SAVE-VIEW;
END-VIEW
```

*First, the `DEFINE-VIEW VS` command creates an empty view schema with the identifier VS. We then define the virtual classes **ALU** and **LogicUnits** (Figure 3.b) and integrate them into GS (Figure 3.c). **LogicUnits** is added to the view with the command `ADD-CLASS(`LogicUnits`)`. Then the three classes of the base schema are added to VS using the command `ADD-VIEW-SCHEMA(BS)` (Figure 3.d). When VS is saved, the is-a arcs shown in Figure 3.e are automatically derived by MultiView.*

4 DESIGN VIEWS FOR HIGH-LEVEL SYNTHESIS

4.1 Introduction

In the following, we present some design view examples for behavioral synthesis tools constructed using MultiView (Figure 5). The goal of this section is (1) to demonstrate the

usefulness and power of the view paradigm, (2) to show how MultiView can be used in a typical CAD application, and (3) to present a solution to the tool integration problem for behavioral synthesis systems. The design views specified in this section are defined on the behavioral design object model (composed of an extended control-flow/data-flow graph augmented with state transition graph information and structural binding and of an extended component graph augmented with behavioral binding and floorplan information), a typical design representation for high-level synthesis systems [2]. Due to space limitations, we refer the reader to [26] for a definition of this underlying global CAD schema.

4.2 A Design View For Component Binding

In this section, we discuss the construction of a design view for the (operator) *binding* design task. *Binding* establishes a mapping between the operators in the data flow graph and the hardware units in the component graph that are to implement the operator (Figure 6.a). Binding typically has constraints, such as: (1) every operator in the data flow graph should be bound to exactly one hardware unit, and (2) two operators can be bound only to the same unit if they are mutually exclusive by being in different states.

The binding tool needs information about the operators in the data flow graph, their assigned state, and components available for a given state. The global schema does of course contain this type of information. It is however spread over several graph structures (namely, the state graph, the data flow graph, and the component graph). For the binding view, this information should be grouped together into a table-like structure, such as the binding table shown in Figure 6.a. In addition, there is the requirement that the binding tool should not modify any of these graph structures. For instance, it should not change the state assignment, the connectivity of the data flow graph, or the set of allocated components. The only legal operations for the binding design task are to establish a binding (i.e., to add a row into the table in Figure 6.a) or to undo a binding (i.e., to remove a row from the table in Figure 6.a).

In the following, we show the steps involved in generating a binding view that meets the described characteristics. We start with the initial global schema shown in Figure 6b. Recall that the binding tool should not be allowed to modify the set of allocated components, e.g., the type of components or their connectivity. Therefore, the design view must protect the **Comp** class. In MultiView, this is accomplished by hiding all update functions from the **Comp** class using the following command:

```
class CompB := hide [setcomptype, setconn, getconn, ...  ]  from Comp;
```

The **hide** command generates the virtual class **CompB**. **CompB** has the same object instance set as the **Comp** class, but a restricted type description. The **CompB** class thus limits access to the components in a design. The binding tool can for instance scan the class of available components using the **CompB** class, but it cannot modify individual components. The integration of the **CompB** class into the global schema in Figure 6.b results in the global schema depicted in Figure 6c. (The class integration algorithm is discussed in Section 3.4).

Next, note that the binding tool should not be able to change the state assignment (a task accomplished by a scheduling tool). For this, we use the following command:

```
   class Dfop1 := refine Dfop with
[boundstate():={return(self.instate.statename)};];
```

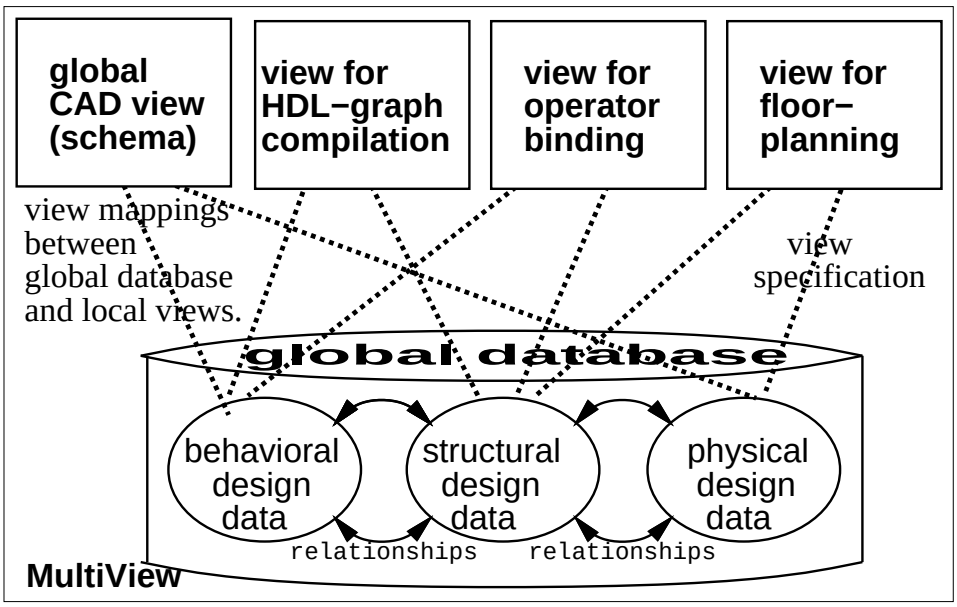


Figure 5: View Support System From the User's Point of View.

The boundstate() function allows the user to retrieve the name of the state associated with a given data flow node, but not the actual state object. Using the new boundstate() function in place of the instate() function will assure that the design tool cannot manipulate the state graph. Class integration of the virtual class **Dfop1** results in the global schema shown in Figure 6.d.

The setbinding() function allows for the binding of an arbitrary data flow node to any component. If, instead, the view definer wants to assure that only legal bindings are being generated, he or she may want to augment the binding design view with application-specific binding functions that include appropriate consistency checks. So we may want to define the **DFop1** class in the following manner:

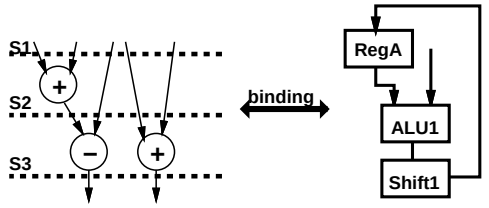
```
class Dfop1 := refine Dfop with
  [boundstate() := { return(self.instate.statename) };
  ssetbinding(c:CompB) := {if compatible(self,c) then setbinding(c);};
  sgetbinding() := {if compatible(self,c) then sgetbinding(c);}; ];
```

The result of integrating **Dfop1** into the global schema is shown in Figure 6.d.

Next, to assure that the design tool cannot manipulate the state graph, we need to remove the instate() function from the design view. In MultiView, this is done using the **hide** operator:

```
class DfopB := hide [instate, setbinding, ...  ]  from Dfop1;
```

This command generates the virtual class **DfopB**, which has the new boundstate() function but not the instate() function in its type description. The integration of **DfopB** into the global schema results in the creation of an intermediate class, which we call **Dfop2**, as shown in Figure 6.e. While the algorithm for class integration is outlined in Section 3.4, the reason for creating these intermediate classes is beyond the scope of this paper [26].



state	op	comp
S1	op1	C1
S1	op2	C3
S2	op3	C1
S2	op4	C2

Binding Table:

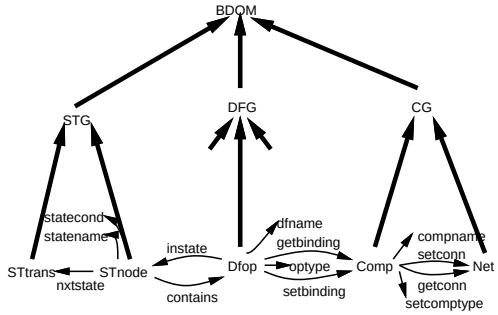
Legal Operations:

setbinding(state,op,comp);
getbinding(state,op,comp);

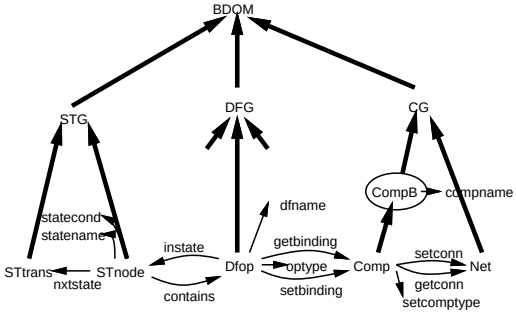
Required Design Data:

operator, assigned state, bound component

(a) The Binding Design Task.

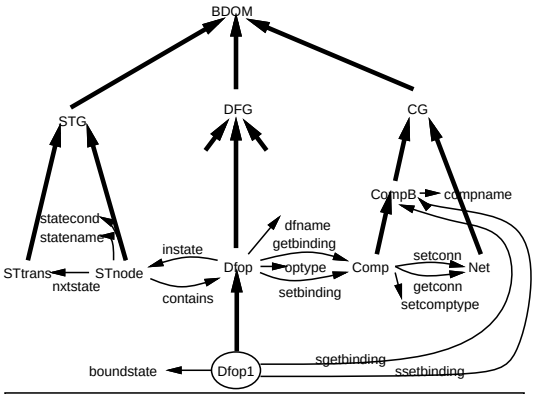


(b) Global Schema of the Behavioral Schema Graph.



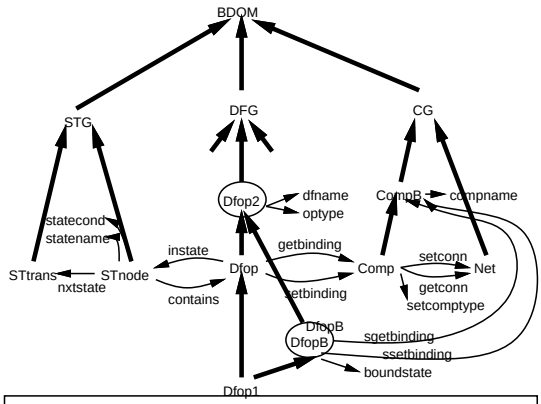

```
class CompB := hide [setcomptype, getconn,... ] from Comp;
```

(c) Hiding Operators From the Comp Class



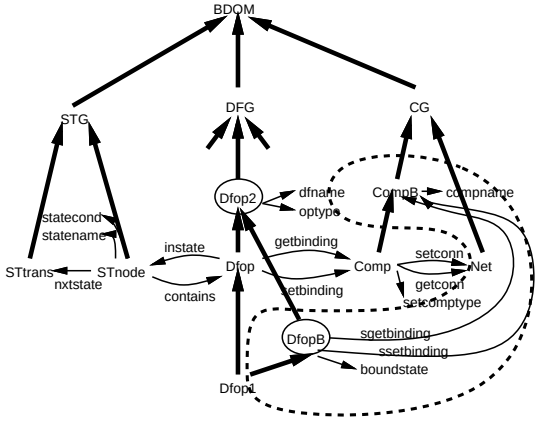

```
class Dfop1 := refine Dfop with
        [ boundstate(String) :=
            {return(self.instate.statename);};
          ssetbinding(c:CompB) :=
              { if compatible(self,c) then setbinding(c);}... ]
```

(d) Adding Refined Operators.




```
class DfopB := hide [instate, setbinding, ... ]  from Dfop1;
```

(e) Removing Illegal Operator from Dfop Class



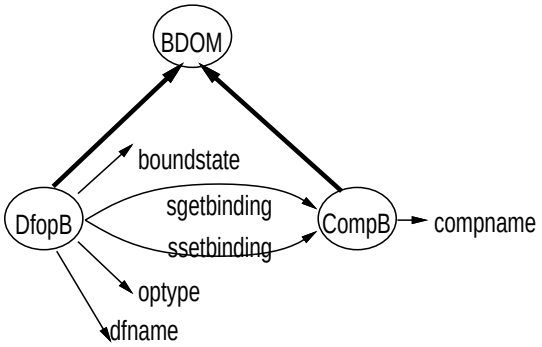

```
ADD−CLASS(DfopB); ADD−CLASS(CompB);
```

(f) Selecting Classes for the View.

Figure 6: The View Specification Steps for the Binding Design View.

```
DEFINE−VIEW BindingView
   class CompB := hide [setcomptype, connto,... ]
         from Comp;
   class Dfop1 := refine Dfop with
         [ boundstate(); ssetbinding(c:CompB); ];
   class DfopB := hide [instate, setbinding, ... ]
         from Dfop1;
   ADD−CLASS(DfopB);
   ADD−CLASS(CompB);
   SAVE−VIEW
END−VIEW
```

(g) View Specification Script of the Binding View



(h) View schema for the BindingView

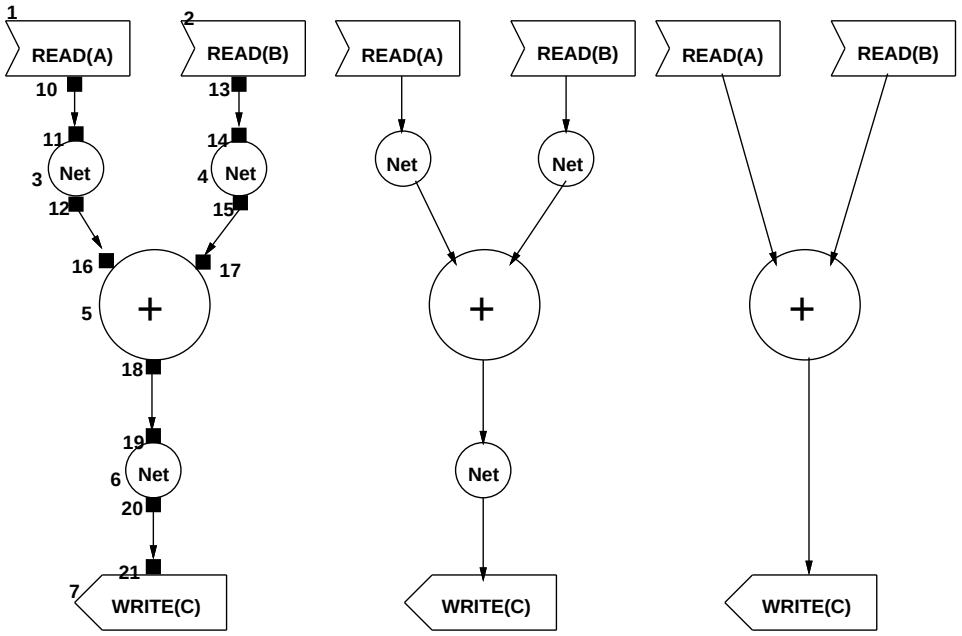
Figure 7: The View Specification of the Binding Design View.

Finally, the appropriate classes must be selected from the global schema to be included in the design view. This is accomplished using the view definition language described in Section 3.5, in this case using the commands `ADD-CLASS (DfopB)` and `ADD-CLASS (CompB)`. The selected view classes **DfopB** and **CompB** are indicated in Figure 6.f by encircling them by a dotted line. The view generation algorithm (Section 3.5) extracts the view classes from the global schema and interconnects them into a view schema. The resulting view schema for the Binding view is depicted in Figure 7.h. For completeness sake, the view specification script to accomplish the specification of the Binding view is listed in Figure 7.h. The Binding view has all features required for the binding design task (and no others). First, the binding triplets can be retrieved and updated using the sgetbinding() and ssetbinding() functions. Second, the list of components can be scanned to determine what components are available, but the allocation of existing components cannot be modified. Third, since each data flow node is assigned to one state, we can retrieve the state associated with a data flow node using the boundstate() function. We can however not modify this state assignment. Clearly, the Binding view protects the design information from being changed in an illegal manner during the binding design task.

4.3 A Design View For Simplying the Data Flow Graph

Some tools, such as the graph compiler and the graph critic, arbitrarily restructure the control/data flow graph [26]. Therefore, an appropriate design view for these tools would simply be the complete set of all behavioral object classes shown Figure 9.a. The allocation tool, on the other hand traverses the data flow graph to determine the number and type of operators. It does not change the structure of the data flow graph, and hence should not have access to the graph-manipulation operators. In addition, the allocation tool may be less interested in the details of how the data flow graph structure is implemented in terms of nets and ports.

This idea of varying levels of detail is best explained using an example. The data flow graph in Figure 8.a represents the design data of the behavioral design specification "C = A + B" in a very detailed manner. It depicts for instance the net objects and the port objects. The net objects represent the data values that flow between data flow node objects. The port objects are (independent) subobjects of data flow nodes and nets that model the interconnection points of these nodes. The other two graphs in Figure 8.b and 8.c

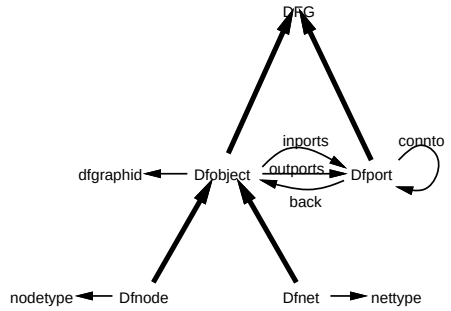


(a) An example DFG graph object.

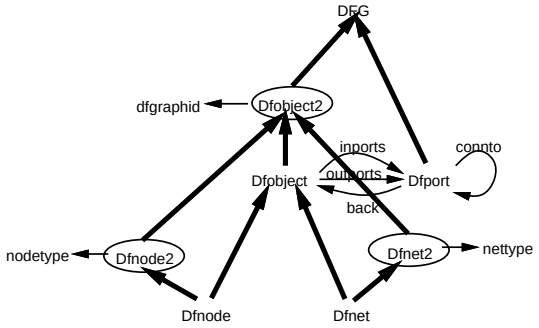
(b) Simplified DFG with ports hidden.

(c) Simplified DFG with nets hidden.

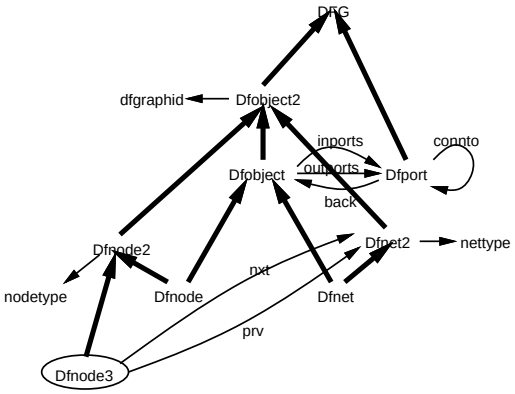
Figure 8: Looking at the Data Flow Graph for the Behavioral Specification "C <= A + B" through Three Different Design Views.



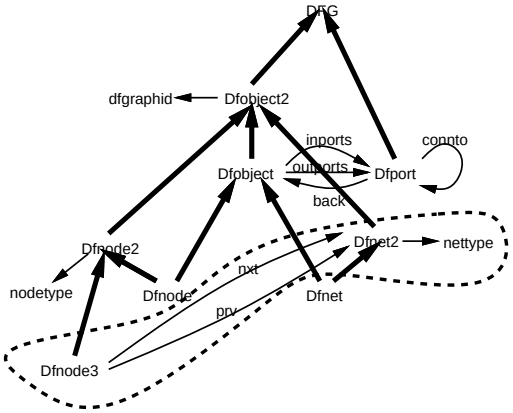
(a) Global Schema for DFG Model (And DFG1 View).



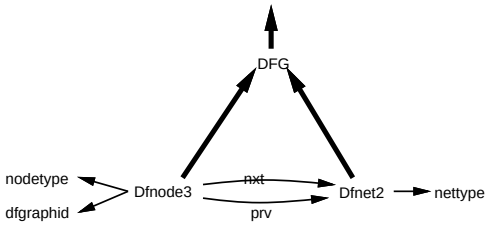
(b) Integration of Virtual Classes into Global Schema.



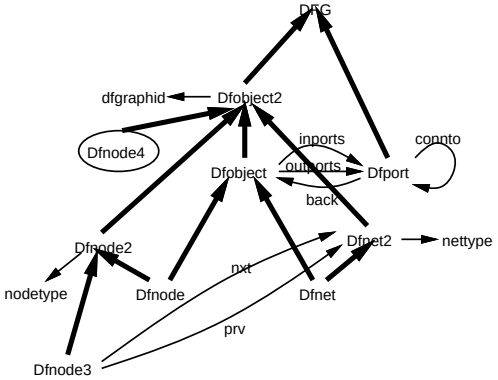
(c) Integration of Virtual Class Dfnode3 into GS..



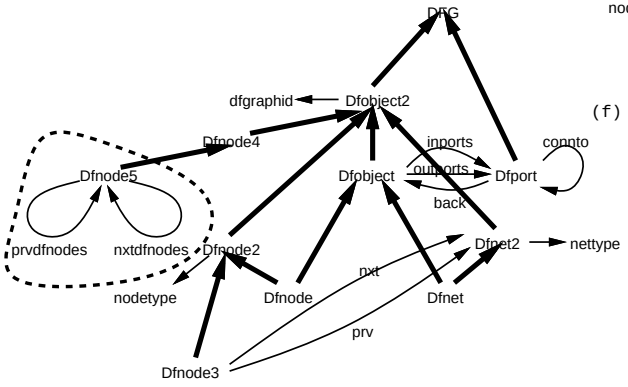
(d) Selecting View Classes for DFG2 View.



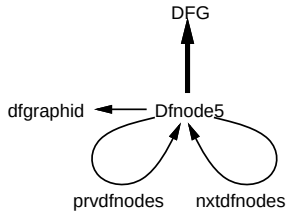
(e) The DFG2 View Schema.



(f) Integration of Virtual Class Dfnode4 into GS..



(g) Integration of Dfnode5 into GS; and Selection of View Classes for the DFG3 View Schema.



(h) The DFG3 View Schema.

Figure 9: The Construction of Three Design View Schemata for the Data Flow Graph Model.

demonstrate how the same data flow graph may be represented using less detail. The data flow graph in Figure 8.b, for instance, no longer represents the ports of the data flow nodes and nets. In the data flow graph in Figure 8.b even the data flow nets are removed. The goal of this section is to develop three design views that correspond to these three different perceptions of the data flow graph model depicted in Figure 8.

As basis of this design view construction we assume the global schema depicted in Figure 8.a. We first specify the design view DFG1 that presents the design data at the level of detail depicted in Figure 8.a. This is equivalent to the behavioral information model captured by the global schema. Hence, DFG1 is simply a subset of the global schema containing all classes related to the data flow graph structure (Figure 8.a). This can be specified by:

```
View Creation Script for the DFG1 View:
DEFINE-VIEW DFG1
   ADD-SCHEMA (DFG);
SAVE-VIEW;
END-VIEW
```

Next, we want to construct a design view (called DFG2) that presents the design data at the level of detail depicted in Figure 8.b. Since the example data flow graph in Figure 8.b does not model the ports, we must remove the **Dfport** class from the design view DFG2. This also includes the hiding of all references to the **Dfport** class. Since the **Dfobject** class has references to the **Dfport** class, its subclasses **Dfnode** and **Dfnet** also inherit these references. Therefore, the two functions inports() and outports() have to be removed from all three classes. In MultiView, this can be achieved by a macro-operator that works on a complete subgraph of the schema rather than on an individual class (see [26]). For this purpose, we use the **hide** macro-operator as follows:

```
class Dfobject2* := hide* [inports(),outports()] from Dfobject*;
```

This query generates three virtual classes **Dfobject2**, **Dfnode2**, and **Dfnet2**, namely, one for each of the classes in the subschema graph rooted at the **Dfobject** class. The result of integrating these three classes into the global schema is shown in Figure 8.b.

By removing the port information from the view schema, we would also remove the information necessary to retrieve the connectivity between data flow nodes and nets. Since this connectivity between data flow nodes and nets must be maintained, we define a function that composes the outports() and inports() functions to calculate the desired connectivity information.

```
class Dfnode3 := refine Dfnode2 with
   [nxt(Dfnet2) :={ return(self.outports.connto.back); };
   prv(Dfnet2) :={ return(self.inports.connto.back); }; ];
```

This **refine** query constructs a virtual class **Dfnode3**, which has the two additional functions nxt() and prv() in addition to the type description of its source class **Dfnode2**. **Dfnode3** is integrated into the global schema by placing it directly below its source class (Figure 9c).

The view DFG2 is now constructed by selecting the view classes **Dfnode3** and **Dfnet2** indicated in Figure 9.d by encircling them with a dotted line. MultiView extracts these view classes from the global schema and interconnects them using generalization relationships into the view DFG2 (Figure 9.e).

```
View Creation Script for the DFG2 View:
DEFINE-VIEW DFG2
   class Dfobject2* := hide* [inports(),outports()] from Dfobject*;
   class Dfnode3 := refine Dfnode2 with [nxt(Dfnet2),prv(Dfnet2)];
   ADD-CLASS (Dfnode3);
   ADD-CLASS (Dfnet2);
SAVE-VIEW;
END-VIEW
```

Next, we construct a design view that presents the design data at the level of detail depicted in Figure 8.c. Since the example data flow graph does not model port nor net objects, we must remove information related to the port and net classes from the design view. Note however that the net objects serve as interconnection points between two data flow nodes. Hence, the connectivity information represented by net nodes must be incorporated into appropriate retrieval functions. This can be accomplished as follows:

```
class Dfnode5 := refine Dfnode3 with
   [nxtdfnodes(Dfnode5) := {return ((cast Dfnode5)
      self.outports.connto.back.outports.connto.back);};
   prvdfnodes(Dfnode5) := {return ((cast Dfnode5)
      self.inports.connto.back.inports.connto.back);};
   ];
```

The virtual class **Dfnode5** still has access to all functions in the type description of Dfnode3, in particular, to the nxt() and prv() functions that reference the **Dfnet** class. Therefore, we'll take the following approach for creating the desired design view:

```
View Creation Script for the DFG3 View:
DEFINE-VIEW DFG3
   class Dfnode4 := select from Dfobject2 where (self in Dfnode);
   class Dfnode5 := refine Dfnode4 with
[nxtdfnodes(Dfnode5),prvdfnodes(Dfnode5)];
   ADD-CLASS (Dfnode5);
   SAVE-VIEW;
END-VIEW
```

The view specification script for design view DFG3 first generates the virtual class **Dfnode4**, which contains all object instances that belong to the **Dfnode** class. The **Dfnode4** class does however have a limited type description, not allowing access to most functions defined for the original **Dfnode** class. The integration of **Dfnode4** into the global schema is demonstrated in Figure 9.f. Next, the view specification script generates the virtual class **Dfnode5**. **Dfnode5** has the desired functions that indicate the connectivity information between two data flow nodes (while hiding the intermediate net and port objects). MultiView integrates **Dfnode5** into the global schema by making it a direct subclass of its source class **Dfnode4** (Figure 9.g). Lastly, the view specification script adds the virtual class **Dfnode5** to the design view DFG3. The selection of view classes is graphically indicated in Figure 9.g using the dotted line. Figure 9.h depicts the third view schema DFG3.

In summary, we have demonstrated the creation of three design views DFG1, DFG2 and DFG3. These three design views hide different levels of detail from the complex data flow graph model. In particular, for the example graph presented in Figure 8, the design views DFG1, DFG2 and DFG3 represent the design data using the data flow graph model on the left-hand side, in the middle, on the right-hand side of the figure, respectively.

Due to space limitations, the reader is referred to [26] for the specification of design views for other design tasks, such as scheduling, allocation, and floorplanning.

5 EVALUATION OF THE DESIGN VIEW APPROACH

The view-based database approach offers all the advantages of a centralized database approach, like, for instance, the integration of diverse design information into one model, integrity control, controlled access to shared data, and the possibility for incremental update. Furthermore, the view-based approach offers additional advantages, such as robustness and flexibility. *Robustness* of the CAD system is achieved by shielding tools from changes in the global data model. As long as the view on which a particular tool operates is maintained, the tool is not affected by changes to the global data model. *Flexibility* of the CAD system is achieved since new customized views, i.e., tool interfaces, can be created rapidly. Hence, new tools can be easily added to the system by simply defining a new view (or possibly using an existing one). Existing tools can work with these new tools through the database without having to develop an additional interface to these new tools (the latter would be required in a design environment with direct tool communication).

Additional advantages of tool integration using design views have been demonstrated by the design view examples discussed in the previous section and elsewhere [26]. A summary of these is given next:

- design views can filter different levels of detail of the otherwise complex design information (e.g., hide complex timing constraints if irrelevant for a design task);
- design views can virtually restructure the design representation graphs so as to narrow the gap between the tool's local model and the global model (such as the complexity of the data flow nets and ports);
- design views can simplify application-specific processing by augmenting the view with customized functions (such as adding move-horizontal() and move-vertical() functions to a floorplanning view);
- design views can increase the level of data consistency by incorporating consistency checks directly into the set of legal access and update functions of the view (such as adding a customized update function for the pins of components that assure the adjacency of the pin with the component position);
- design views can precompile information that is frequently needed by the users of the view by maintaining derived attributes (such as adding the absolute-position() attribute of pins calculated based on the relative pin positions and the position of the component);
- design views can assure the correct update of designs by preparing appropriate update functions and associating them with the view, while hiding all illegal operators from the view.

A number of observations arose from our experience of defining design views for the different behavioral synthesis tasks. The key observations are listed below.

- We found that the specification of design views is relatively simple when using the view definition language. It requires of course an understanding of (a) the global model of the design information and (b) the information needs of the particular design task.
- The generation of design views is much less labor-intensive compared to manually and in an ad-hoc fashion having to implement a tool interface and/or data file translators.
- We found that MultiView was sufficiently expressive to handle the specification of the design views for all behavioral synthesis tasks that we explored.
- The type-manipulating object algebra operators, such as **hide** and **refine**, were more frequently used than the set-manipulating ones, such as **select** and **union**. A reason for this may be that the base schema already represents an appropriate classification of the design objects into meaningful classes.
- Theoretically, the global schema could explode in size with the addition of many new views. We found that class explosion was not a problem for the CAD example views we studied. First, there generally is only a limited number of different views of interest for a given application. Secondly, different views often use subparts of the global schema in the same manner. For instance, both the allocation and binding design tools use the subschema about the behavioral design data for read-only purposes.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a solution to the problems caused by utilizing a global design database to achieve the much needed integration of diverse design information in a CAD system. We have proposed to utilize the object-oriented view methodology, called MultiView, for specifying customized tool interfaces (design views) on the CAD database. A design view contains a subset of relevant information from the global database organized in a fashion most suitable to the needs of a particular design tool. MultiView automatically maintains the mapping between the global data model and local design views, thus freeing individual design tools from this burden. In this paper, we have also given numerous examples that demonstrate MultiView and its advantages for typical tasks in high-level synthesis. Contributions of this paper to the CAD field can be summarized as follows:

- the identification of the problem of a conflict between data unification achieved by a central database and data customization required by individual design tools;
- the recognition that object-oriented database views can be used to address this problem (put differently, the proposal of a novel approach towards addressing the tool integration problem),
- the demonstration of the utility of MultiView for CAD applications by defining example design views for particular design tasks, and
- the refinement of the architecture of a CAD framework to include a component for *view support*.

Finally, we have (in a small way) contributed to the area of object-oriented views by demonstrating their usefulness for a particular application domain.

To our knowledge, this is the first time that the concept of database views has been applied to CAD applications. We think that the introduction of the view methodology to the CAD field will revolutionize the currently clumsy mechanism of tool integration using pairs of file translators. The creation of design views on the CAD object model will result in a design environment that overcomes the problems caused by a central database, such as, limited extensibility and generality of the global model. We expect this to be an important first step towards the development of more powerful CAD environments based on this flexible view approach.

While much has been accomplished, this work has opened many more avenues for further research. While relational databases support some form of views, none of the commercial object-oriented database systems (nor the prototypes available at universities) provide the necessary functionality for view support as of today. Therefore, we are currently working on an implementation of MultiView on top of the object-oriented database GemStone[2]. Based on our experience with modeling typical design views using the view specification language provided by MultiView, we feel strongly that object-oriented views will prove to be a viable approach towards customized tool integration support.

Once the MultiView prototype has been completed, there are a number of challenging issues that need to be studied. For instance, we want to determine the ease of view construction and the modeling power of MultiView for all desirable design tasks. More importantly, however, for the final acceptance of such a system by the CAD industry, comparison studies among different tool integration approaches, such as (1) using file translators, (2) using one central database together with extracting routines, and (3) using the view-based database will have to be performed. Two of the most important measures of this comparison will be (1) the time spend to develop new design tools and (2) the performance of typical design tasks executing in these different CAD systems.

Another avenue of future research concerns the exploration of the scope of the "reorganization" power of the view support system. It is a challenging open problem whether a view-based approach could be used to automatically maintain the mapping between drastically different representation paradigms, e.g., between the graph structures used in high-level synthesis versus the special-purpose hash-based matrix representations of a netlist used for simulation [27].

Acknowledgements. I want to thank Lubomir Bic and Daniel D. Gajski for providing me with advice and encouragement during earlier stages of this work.

References

[1] Abiteboul, S., and Bonner, A., "Objects and Views," in *Proc. SIGMOD*, May 1991, pp. 238 – 247.

[2] Afsarmanesh, H., Brotoatmodjo, E., Ryeon, K. J., Parker, A. C., The EVE VLSI Information Management Environment, *IEEE Int. Conf. on CAD*, pp. 384 - 387, 1989.

[2] GemStone is a register trademark of the Servio Corporation.

[3] Allen, W., Rosenthal, D., and Fiduk, K., "The MCC CAD Framework Methodology Management System," *DAC'91*, pp. 694 – 698.

[4] Bancilhon and W. Kim, "Object-Oriented Database Systems: In Transition," *SIGMOD RECORD,* Vol. 19, No. 4, Dec. 1990, pp. 49 – 53.

[5] Baer, J. L., Liem, M. C., et al., A Notation for Describing Multiple Views of VLSI Circuits, *DAC'88*, pp. 102 – 107.

[6] Bingley, P., and P. Van der Wolf, A Design Platform for the NELSIS CAD Framework, *DAC'90*, 146 - 149.

[7] Blackburn, R. L., Thomas, D. E., and Koenig, P.M., Linking the Behavioral and Structural Domains of Representation for Digital System Design, *IEEE Trans. on CAD*, vol. CAD-6, No. 1, Jan. 1987.

[8] Cattell, R. G. G., *Object Data Management*, Addison-Wesley, 1992.

[9] CAD Framework Initiative, Panel Discussion, *29th ACM/IEEE Design Automation Conf. (DAC'92)*, Anaheim, California, June 1992.

[10] Chiueh, T.-C., and Katz, R.H., Intelligent VLSI Design Object Management, *EDAC'92*, pp. 410 – 414, Feb. 1992.

[11] Daniell, J. and Director, S., An Object Oriented Approach to CAD Tool Control, *Proc. 26th Design Automation Conference*, pp. 197 – 202, 1989.

[12] Gajski, D. D., Dutt, D. N., Wu, A. C.-H., and Lin, S. Y.-L., *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Press, 1992.

[13] Graham, A., "The CAD Framework Initiative Standards Progress Towards First Publication at Year End", *IEEE DATC Newsl. on Design Automation*, Sp. 1992, pp. 13–21.

[14] Gupta, R., Cheng, W. H., Gupta R., Hardonag, I. and Breuer, M. A.. An Object-Oriented VLSI CAD Framework, *IEEE Computer*, vol. 22, no. 5, 28 – 37, May 1989.

[15] Hamer, P. and Treffers, M., A Data Flow Based Architecture for CAD Frameworks, *Proc. IEEE Internat. Conf. on Computer-Aided Design*, pp. 482 – 485, 1990.

[16] Harrison, D. S., Moore, P., Spickelmier, R. L., and Newton, A. R., Data Management and Graphics Editing in the Berkeley Design Environment, pp. 24 – 27, *ICCAD'86.*

[17] Heijenga, W., Jasnoch, U., and Radeke, E., "DaDaMo: A Conceptual Data Model for Electronic Design Applications", *EDAC'92*, pp. 394 – 398.

[18] Heiler, S., and Zdonik, S. B., "Object views: Extending the vision", in *Proc. IEEE Data Eng. Conf.*, Feb. 1990, pp. 86 - 93.

[19] Knapp, D. W., and A. C. Parker, A unified representation for design information, In *Proc. CHDL-85,* Elsvier, 1985.

[20] Lanneer, D., et al., An Object-oriented framework supporting the full high-level synthesis trajectory, *CHDL'91*, pp. 281 – 300, 1991.

[21] Miller, J., Strauss, J., and Rammig, F., "Integration of a CHDL into an Engineering Environment," *CHDL'90*, pp. 157 – 166.

[22] Mueller, W, and Rammig, F., "ODICE: Object-Oriented Hardware Description in CAD Environment," *CHDL'90*, pp. 19 – 34.

[23] Rammig, F., (editor), IFIP WG 10.2, Workshop on Electronic Design Automation Frameworks, Nov. 1990.

[24] Rundensteiner, E. A., "*MultiView:* A Methodology for Supporting Multiple Views in Object-Oriented Databases", *Int. Conf. on Very Large Data Bases*, 1992, pp. 187-198.

[25] Rundensteiner, E. A., and Bic, L., "Set Operations in Object-Based Data Models", in *IEEE Transaction on Data and Knowledge Eng.,* vol. 4, issue 4, August 1992, pp. 382 – 398.

[26] Rundensteiner, E. A., "Object-Oriented Views: A Novel Approach to Tool Integration in Design Environments," Dissertation, Info. and Computer Science Dept., Univ. of California, Irvine, Fall 1992.

[27] Sangiovanni-Vincentelli, A., Univ. of Michigan, Ann Arbor, Private Communication, Oct. 1992.

[28] Scholl, M. H., Laasch, C. and Tresch, M., "Updatable Views in Object-Oriented Databases," in *Proc. 2nd DOOD Conf.*, Germany, Dec. 1991.

[29] VanEijndhoven, J. T. J., and Stok, L, "A Data Flow Graph Exchange Standard," *EDAC'92*, pp. 193 – 199, 1992.

[30] Wu, A. C. H., Hadley, T. S., and Gajski, D. D., An Efficient Multi-View Design Model for Real-Time Interactive Synthesis, *ICCAD'92*, pp. 328 – 331.