THE DATA MODEL OF THE STAR FRAMEWORK

por

Flávio Rech Wagner

RP nº 167 NOVEMBRO/91

"Trabalho realizado com o apoio do CNPq".

SABi
UFRGS
05234786

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Av. Bento Gonçalves, 9500 - Agronomia
91501 - Porto Alegre - RS - BRASIL
Telefones: (0512) 36-8399/39-1355 - Ramal 6161
Telex: (051) 2680 - CCUF - BR
FAX: (0512) 24-4164
E-mail: PGCC@VORTEX.UFRGS.BR
Correspondência: UFRGS/CPGCC
Caixa Postal 15064
91501 - Porto Alegre - RS - BRASIL

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Editor: Ricardo Augusto da Luz Reis (interino)

Informática - SBC
CAD: Sistemas digitais
Ambiente: Projeto
Modelos: Dados

CNPq 1.03.03.00-6

UFRGS INSTITUTO DE INFORMÁTICA BIBLIOTECA		
Nº CHAMADA FL 2150		Nº REG.: 36126 DATA: 24/02/92
ORIGEM: D	DATA: 27/11/91	PREÇO: Cr$ 10.000,00
FUNDO: CPGCC	FORN.: CPGCC	

UFRGS

Reitor: Prof. TUISKON DICK
Pró-Reitor de Pesquisa e Pós-Graduação: Prof. ABÍLIO BAETA NEVES
Coordenador do CPGCC: Prof. Ricardo A. da L. Reis
Comissão Coordenadora do CPGCC: Prof. Carlos Alberto Heuser
Prof. Clesio Saraiva dos Santos
Profa. Ingrid Jansch Pôrto
Prof. José Mauro V. de Castilho
Prof. Ricardo A. da L. Reis
Prof. Sergio Bampi
Bibliotecária CPGCC/II: Margarida Buchmann

Abstract

This report presents the data model for the design framework STAR. STAR is an open framework for the design of electronic circuits and systems, that provides powerful facilities for data and design management, as well as for cooperation among designers. The data model is the basis for preserving data consistency throughout the design process. It allows different organizations for the multiple representations of a design object that are created during this process. This flexibility allows the framework to support various data and design management schemes. The data model is specified through the Plasma language, a semi-formal system specially developed for this task.

Keywords
Electronic design automation. Design frameworks. Data representation model.

Resumo

Este relatório apresenta o modelo de dados do ambiente de projeto STAR. STAR é um ambiente aberto para o projeto de sistemas e circuitos eletrônicos, que oferece recursos poderosos para a gerência de dados e de projeto, assim como para a cooperação entre projetistas. O modelo de dados é a base para a preservação da consistência dos dados ao longo do processo de projeto. Ele permite diferentes organizações para as múltiplas representações de um objeto de projeto criadas ao longo deste processo. Esta flexibilidade permite que o ambiente suporte vários esquemas de gerência de dados e de projeto. O modelo de dados é especificado através da linguagem Plasma, um semi-formalismo desenvolvido especialmente para esta tarefa.

Palavras-chave
Automação do projeto de sistemas eletrônicos. Ambientes de projeto. Modelo de representação de dados.

Contents

List of Figures

1 Introduction

Design frameworks aim at the integration of tools so as to guarantee the overall consistency of the process of designing circuits and systems. They are typically based on three main interfaces. The data interface, which allows access to a common data base, uses a uniform data model for representing complex objects at various abstraction levels. The user interface offers graphical facilities that make possible the uniformization of the user interaction style. The system interface, finally, offers facilities for tool execution that are typically found in operating systems. Examples of frameworks that partially or totally support these features are Oct [1], from Berkeley, FACE [2], from GE, Cadweld [3], from Carnegie-Mellon, and CWS [4], from the Cadlab.

The development of the STAR framework [5] is a joint effort of the UFRGS and the IBM Rio Scientific Center which is based on previous experience of these groups in the field of design frameworks (the AMPLO environment [6], at the UFRGS, and the GARDEN environment [7], at IBM). The STAR framework will support the most important features expected from systems effectively open to the integration of tools aimed at various applications, architectures, and technologies. STAR will be based on a data model which is derived from the GARDEN model and which has been shown to support superior concepts with regard to other frameworks [8]. The STAR framework will offer special facilities for data and design methodology management and for cooperation between designers.

This report presents the STAR data model in its entirety, in a semi-formal way, by using an ad-hoc specification language. The report describes the objects, their attributes, and the relationships between them, but it does not describe the operational part of the model, i.e. the functions that can be applied for manipulating these objects (functions for finding, creating, removing, or modifying objects, as well as functions for navigating through the objects according to the relationships between them). This operational description will be the subject of a following report. An overall description of the STAR framework can be found elsewhere [5].

This report is organized as follows. In Section 2, the semi-formal specification language Plasma is presented. Section 3 then completely presents the STAR data model by using Plasma. Section 4 discusses issues related to the configuration of objects.

2 The specification language Plasma

Plasma is an ad-hoc specification language, specially designed for the specification of the STAR data model [1]. Plasma supports abstraction concepts that can be mapped to well-known modelling concepts like aggregation, generalization, and association. However, since the only goal of Plasma is to serve as a means of unambiguosly specifying the STAR data model, there is no intent of obtaining a general purpose specification language. It is not argued that Plasma is a formal language, neither that its mechanisms are complete, orthogonal, or even consistent with each other. They will be presented and explained through examples, not through formal syntactical and semantic description. These examples, although coming from the digital systems area, have no relationship to the STAR concepts.

2.1 Notation

The following notation is used in the specification of the Plasma language:

1. Words beginning with a lower case are reserved words;
2. Words beginning with an upper case are user-defined words, like object names;
3. [ ] means an optional item
4. { } means an iteration – zero or more occurrences of the item
5. $\{\ \}_a^b$ means a limited iteration – minimum of a and maximum of b occurrences of the item

2.2 Objects and attributes

Elements of a Plasma description are objects and attributes. Objects are the main components of a data model, like entities in a E/R diagram. Attributes serve to model properties of objects. Attributes do not have autonomous existence, as opposed to objects. Deciding whether to model a data model element as an object or as an attribute is a matter of modelling purpose. In the following STAR data model specification, for instance, Ports (interface signals of hardware modules) are modelled as attributes of design objects, although one could perfectly describe them as objects. However, attributes can be inherited by objects from other objects, and this property was decisive in the modelling choice for Ports.

Objects and attributes can be complex elements, that can be formed by composition of sub-elements (aggregations).

Object and attribute declarations in Plasma are like type definitions in programming languages. Many instances of an object or attribute type can exist in a concrete system represented through the data model specified in Plasma.

[1] Plasma is the state of the material which constitutes a star during its birth phase.

2.3 Aggregations

An element can be an aggregation of other objects and attributes. Attributes can be single or composite. Component objects are specified through the *contains* sentence, while attributes are declared by the *has* sentence. An element declaration can have any number of such sentences. There must be one sentence for each component object or attribute type. Figure 1 illustrates the declaration of components and attributes. Adder and Subtractor are components of the object ALU, while InterfaceEnvelope and BitWidth are its attributes. InterfaceEnvelope is a composite attribute.

```
ALU contains Adder
    contains Subtractor
    has InterfaceEnvelope
    has BitWidth
InterfaceEnvelope has HorizDimension
                  has VertDimension
```

Figure 1: Component and attribute sentences

Sentences can specify optional attributes and components, as well as multiple attributes and components of the same type, as shown in Figure 2. The object ALU contains many components of type InterfaceSignal and an optional component of type OverflowFF. It also has many optional attributes of type TimingParameter. The attribute InterfaceEnvelope has many sub-attributes of type Vertice and optional sub-attributes of types HorizDimension and VertDimension.

```
ALU contains { InterfaceSignal }_1^n
    [ contains OverflowFF ]
    has { TimingParameter }
    has InterfaceEnvelope
InterfaceEnvelope has { Vertice }_1^n
                  [ has HorizDimension ]
                  [ has VertDimension ]
```

Figure 2: Optional and multiple components and attributes

2.4 Generalization and association

An object can be a generalization of other objects, called sub-objects, and also an association of these same sub-objects. As a generalization, the object has attributes that are inherited by the sub-objects. Inherited attributes are listed in an *inheritance clause*, specified within the symbols {{ and }}. The attributes defined in such a

clause do not need to be defined through a *has* sentence. There are two different types of inheritances, that will be discussed later on. As an association, the object has attributes that are not inherited by the sub-objects.

Generalization objects in Plasma can be versioned (see subsection 2.9 on object versioning). Objects Y_i generalized by X can inherit different attribute values from different versions of X.

In the example in Figure 3, ArithmElement is a generalization and an association of objects of type Adder. Attribute BitWidth of ArithmElement is inherited by all objects of type Adder, while attribute Designer is not.

```
ArithmElement has Designer
                    is generalization of Adder
                    {{ inherited BitWidth }}
```

Figure 3: Generalization and association

An object can be a generalization of different sets of sub-objects simultaneously. Simultaneous generalizations can be specified through a single *generalization* sentence. In Figure 4, both object types Adder and Subtractor inherit the attribute BitWidth of ArithmElement.

```
ArithmElement is generalization of { Adder ; Subtractor }
                    {{ inherited BitWidth }}
```

Figure 4: Generalization of different sets

Sub-objects of a generalization / association are identified by a hierarchical, prefixed naming convention. At each level of the hierarchy, an object is identified by preceding its name by the names of all its ascendants up in the hierarchy. In the example in Figure 5, suppose an ALU object with Name *ALU1*. A sub-object ALU-Version of this ALU, with Name *V1*, has a composed Name *ALU1.V1*. A sub-object ALU-Revision of this ALU-Version, with Name *R1*, has a composed Name *ALU1.V1.R1*.

```
ALU has Name
    is generalization of ALU-Version
ALU-Version has Name
            is generalization of ALU-Revision
```

Figure 5: Naming convention

2.5 References and scope of definition

An aggregation can also contain references to other elements. References can be done to either objects or attributes. A reference is like a pointer in a programming language. It does not define a new object or attribute, but states that an element has some sort of relationship with another element. A reference is specified through the *reference* sentence. It contains a label (the reference identifier), followed by a colon and the name of the referenced element. References are always considered as attributes of elements. An attribute of another object can be referred to by a prefixed notation. In the example of Figure 6, Add1 is an attribute of ALU which makes reference to another object Adder, while Cell1 contains a reference to the attribute PortLayer of Cell2.

```
ALU has reference Add1 : Adder
Cell1 has reference RefLayer : Cell2 . PortLayer
```

Figure 6: References

An attribute which is defined in an object X is also known in all sub-objects that are hierarchically contained in X. In Figure 7, FlipFlop makes a reference to an attribute DesignStyle, which is defined in DataPath, that hierarchically contains FlipFlop, while a reference to ClockScheme of ControlBlock, which is unrelated to FlipFlop, must be done through a prefixed notation.

```
DataPath has DesignStyle
        contains { Register }_1^n
Register contains { FlipFlop }_1^n
ControlBlock has ClockScheme
FlipFlop has reference FFStyle : DesignStyle
        has reference CKScheme : ControlBlock . ClockScheme
```

Figure 7: References and scope of definition

2.6 Alternative definitions

Sometimes it is desirable, for modelling purposes, to introduce an element which is a synonim for any of a list of alternative elements. This can be done by the *alias* sentence. In the example shown in Figure 8, FlipFlop can be replaced by any of the objects JK-FF, D-FF, and RS-FF.

Register contains { FlipFlop }$_1^n$
FlipFlop is { JK-FF / D-FF / RS-FF }

Figure 8: Alternative definitions

2.7 Inheritance types

Sub-objects in a *generalization* sentence can inherit attributes of the generalization objects according to two different inheritance rules.

It must be noted that generalizations can be applied in a cascaded way, as shown in Figure 9. MemoryElement is said to be an ascendant of Register, while MemoryElement and Register are ascendants of D-Register. Attributes can be inherited from all ascendants of an object. D-Register is said to be a descendant of both MemoryElement and Register.

MemoryElement is generalization of Register
Register is generalization of D-Register

Figure 9: Cascaded generalizations

In the *default inheritance*, inherited properties (attribute plus value) are valid only if they are not overriden in the definition of the descendant. In the example in Figure 10, the value of ClockFrequency can be determined by MemoryElement and thus inherited by Register and D-Register. This value can be however redefined by either Register or D-Register.

MemoryElement is generalization of Register
{{ inherited ClockFrequency }}
Register is generalization of D-register

Figure 10: Default inheritance

In the *strict inheritance*, all inherited properties must exist and be valid for each descendant. The value of an inherited attribute cannot be defined in a sub-object if it has been already defined at an upper level. This is illustrated in Figure 11, where D-Register inherits SetUpTime from Register and ClockFrequency from MemoryElement. A different value can be assigned to SetUpTime in each object of type D-Register. If, however, the value of SetUpTime is already defined in a Register object R, then it will be inherited by all sub-objects of type D-Register of R. The value of ClockFrequency, if already defined in a MemoryElement object, cannot be changed by objects of type Register or D-Register.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

```
MemoryElement is generalization of Register
                    {{ strict inherited ClockFrequency }}
Register is generalization of D-Register
                {{ strict inherited SetUpTime }}
```

Figure 11: Strict inheritance

2.8 Data types

Non-complex attributes (i.e attributes that are not specified as aggregations of other attributes) must have a data type associated with them. Data types can be primitive or composite. Primitive data types in Plasma are *integer*, *bit-vector*, *file*, *string*, and *time*. Composite data types are *sets*, *arrays*, and *records*. The data type of a non-complex attribute must be defined in the *has* sentence. Examples are given in Figure 12.

```
RegisterFile has BitWidth : integer
             has Name : string
             has RegisterSet : array (1 to 8) of bit-vector
             has DataInput : record of [ SetUpTime : time;
                                         Data : bit-vector ]
```

Figure 12: Data types

A non-complex attribute can also have a constant value of any of the primitive or composite data types, as shown in Figure 13.

```
RegisterFile has BitWidth : 16 (an integer value)
             has DesignStyle : standard-cell (a string value)
```

Figure 13: Data types with constant values

As another option, a data type can be implicitly defined by enumeration of values or by subseting of one of the above mentioned primitive data types, as illustrated in Figure 14.

2.9 Versionable objects

Plasma allows the specification of versionable objects, a fundamental requirement for data models in engineering applications. This property can be attached either to objects, through a *is versionable* sentence, or to single attributes of objects, through

```
Port has Direction : { in / out / inout } (enumeration of values
                                          of type string )
     has Delay : { 10 ns to 20 ns } (subseting of type time)
     has Value : bit-vector {8} (a bit-vector of 8 bits)
```

Figure 14: Enumeration and subseting

a *versionable* qualifier. In the case of a versionable object, any change in the value of any object attribute will imply the creation of a new object version. In the case of versionable attributes, a new object version is created only when the value of a versionable attribute of the object changes. In the example shown in Figure 15, changes in the value of Delay or ClockFrequency will create a new version of a Register object.

```
Register has Delay
         has ClockFrequency
         is versionable
```

Figure 15: Versionable objects

In the next example, in Figure 16, only changes in the value of Delay will create a new version of Register.

```
Register has Delay versionable
         has ClockFrequency
```

Figure 16: Versionable attributes

The combination of the *versionable* qualifier with sets of attributes is interpreted as follows: not only the values of the attributes are versionable, but also the composition of the set of attributes is versionable. In the example of Figure 17, ALU can have many attributes of type TimingParameter. A new version of ALU is created when: a) the value of ParamName or ParamValue of any of the TimingParameters changes, and b) a new attribute of type TimingParameter is created for ALU or an already existing attribute is deleted.

This interpretation for versionable sets of attributes also holds when a whole object is versionable and it contains such sets, as in the case of object Register.

Versioning can also be applied to sets of components in composite objects, as for the object Shifter. In this case, a new version of Shifter is created when a sub-object Shifter-Slice is created or deleted.

ALU has { TimingParameter } versionable
TimingParameter has ParamName : string
has ParamValue : time
Register has { TimingParameter }
is versionable
Shifter contains { Shifter-Slice }$_n^!$ versionable

Figure 17: Versioning sets of attributes

3 The STAR data model

In this section all objects of the STAR data model are specified through the Plasma language. Sometimes during the specification, it will be necessary, in order to explain properties of the objects, to make references to other objects that will be defined only in a later subsection.

This specification does not contain the operational part of the model, i.e. the functions that can be applied for manipulating objects and navigating through the objects according to the relationships between them.

3.1 Repository

The Repository, whose definition is shown in Figure 18, is the collection of all objects in the data base. It is composed of Libraries, where the design objects are stored, and of Processes, that contain technology-related information.

```
Repository contains { Process }
           contains { Library }_1^n
```

Figure 18: Repository

3.2 Processes

A Process is a collection of technology-related information which is stored in one or several TechFiles as bit-strings, as shown in Figure 19. The contents of the TechFiles are directly handled by the design tools.

A Process can optionally refer to many includable Processes, that are other Processes with compatible technologies. This information can be used for the configuration of composite object descriptions. A rule could for instance state that an object X with Process PX can contain a sub-object which is a reference to an object Y with Process PY iff PY is includable in PX.

The set of includable Processes is fixed and cannot be changed in consecutive versions of the Process.

```
Process has Name : string versionable
        contains { TechFile : file versionable }_1^n
        has { reference IncludableProcess : Process }
```

Figure 19: Processes

3.3 Libraries

A Library, defined in Figure 20, is a collection of design objects. A design object is stored in a single Library, so that Libraries are physical, not logical, partitions of objects. A Library can optionally have an associated Process. This Process is then necessarily associated with all design objects of the Library. The design objects can also inherit some attributes (UserFields) of the Library. Attribute inheritance can be of type strict or default.

A Library LX can optionally refer to many associated Libraries $L1$,...,Ln. This information can be used for the configuration of composite object descriptions. A rule could for instance state that an object X from a Library LX can contain a sub-object which is a reference to an object Y from a Library LY iff LY is associated with LX.

The associated Process and the set of associated Libraries are fixed and cannot be changed in consecutive versions of the Library.

```
Library has Name : string versionable
        [ has reference AssociatedProcess : Process ]
        has { reference AssociatedLibrary : Library }
        has { UserField versionable }
        is generalization of Design
          {{ strict inherited AssociatedProcess; UserField }}
          {{ inherited UserField }}
```

Figure 20: Libraries

3.4 Designs

A Design is a single design object, like a microprocessor, an ALU, a register, or a gate. Its specification is shown in Figure 21. A Design inherits the Process of the Library in which it is contained. If such a Process is not specified, a Process can be directly attached to the Design. The associated Process is fixed and cannot be changed in consecutive versions of the Design.

Ports that are common to all representations of a design object can be stored at the Design, which is the root node of a hierarchy of representations (called the Design *control structure*).

Designs are collections of ViewGroups and Views, that inherit some attributes of the Design (Ports among them). UserField inheritance can be of type strict or default, but Port inheritance is always of type strict. The name of any representation of a Design is prefixed by the Design name.

Designs can be parameterized objects. This means that the behavior of instances of a Design (used in Components inside structured Views of other Designs) can depend on the value of one or more Parameters. Actual values cannot be assigned to a Parameter when the Design is declared. These values are assigned only when the

instances are created. All Parameters of a Design are inherited by the ViewGroups and Views under the Design. Parameter inheritance is always of type strict.

```
Design has Name : string versionable
       [ has reference AssociatedProcess : Process ]
       has { Port versionable }
       has { UserField versionable }
       has { Parameter versionable }
       is generalization of { ViewGroup ; View }
         {{ strict inherited Port ; UserField; AssociatedProcess; Parameter }}
         {{ inherited UserField }}
```

Figure 21: Designs

3.5 ViewGroups

A ViewGroup, whose definition is found in Figure 22, is a collection of representations for a design object that have some common properties. These properties can be user- or methodology-defined. An attribute GroupCriteria can store these properties for documentational purposes.

A ViewGroup inherits the Process of the Design or ViewGroup to which it belongs. If such a Process is not specified, a Process can be directly attached to the ViewGroup. The associated Process and the GroupCriteria are fixed and cannot be changed in consecutive versions of the ViewGroup.

A ViewGroup can be further decomposed into other ViewGroups and Views, that inherit some of its attributes. ViewGroups inherit Ports from the Design and/or ViewGroups above them in the control structure. UserField inheritance can be of type strict or default, while Port inheritance is always of type strict. Ports that are common to all representations gathered in a ViewGroup can be stored at this node of the control structure. The name of any representation below a ViewGroup in the control structure is prefixed by the ViewGroup name.

The ViewGroup can add new Parameters to the design object. They will be inherited, in strict mode, by the ViewGroups and Views under this ViewGroup.

3.6 Views

A View is a representation of a design object at a given abstraction level. Its specification is shown in Figure 23. For each View there can be any number of ViewStates, where the concrete design data are stored.

A View inherits the Process of the Design or ViewGroup to which it belongs. If such a Process is not specified, a Process can be directly attached to the View. All

```
ViewGroup has Name : string versionable
        [ has GroupCriteria : string ]
        [ has reference AssociatedProcess : Process ]
        has { Port versionable }
        has { UserField versionable }
        has { Parameter versionable }
        is generalization of { ViewGroup ; View }
          {{ strict inherited Port ; UserField ; AssociatedProcess; Parameter }}
          {{ inherited UserField }}
```

Figure 22: ViewGroups

ViewStates of a View have the same Process. The associated Process is fixed and cannot be changed in consecutive versions of the View.

Views inherit Ports from the Design and/or ViewGroups above them in the control structure. Ports that are common to all representations gathered in a View can be stored at this node of the control structure. Port inheritance towards the ViewStates is always of type strict.

The name of any representation below a View in the control structure is prefixed by the View name.

The View can add new Parameters to the design object. They will be inherited, in strict mode, by all ViewStates under the View.

```
View has Name : string versionable
        [ has reference AssociatedProcess : Process ]
        has { Port versionable }
        has { UserField versionable }
        has { Parameter versionable }
        is { HDLView / LayoutView / MHDView }
```

Figure 23: Views

There are three View types, whose definition can be found in Figure 24. The HDLView is dedicated mainly for behavioral descriptions at high abstraction levels, normally using some hardware description language, such as VHDL. The MHDView (Modular Hierarchical Description View) is used for purely structural descriptions, for instance at the RT, logic, or electrical level. The LayoutView, finally, is oriented for the geometrical description of physical realizations, either the layout of integrated circuits or of printed circuit boards.

```
HDLView is generalization of ViewState
        {{ strict inherited Port ; UserField ; AssociatedProcess; Parameter }}
        {{ inherited UserField }}
LayoutView is generalization of ViewState
        {{ strict inherited Port ; UserField ; AssociatedProcess; Parameter }}
        {{ inherited UserField }}
MHDView is generalization of MHDViewState
        {{ strict inherited Port ; UserField ; AssociatedProcess; Parameter }}
        {{ inherited UserField }}
```

Figure 24: View types

3.7 ViewStates

ViewStates are the nodes of the control structure where concrete design data are stored.

The ViewStates of a given View are organized as a derivation graph, where each ViewState has a number of predecessor nodes and a number of successor nodes. There is no inheritance of attributes between ViewStates. All ViewStates of a graph inherit their attributes directly from the View. The predecessor and successor nodes of a ViewState are not versionable.

Ports can be specified at the ViewState level. They are added to Ports already specified for the design object in the ascendants of the ViewState in the control structure. Parameters cannot be added to the design object at the ViewState level.

```
ViewState has Name : string versionable
          has CreationDate : time versionable
          has { Port versionable }
          has { UserField versionable }
          has { reference PredViewState : ViewState }
          has { reference SuccViewState : ViewState }
          has ViewDescription : file versionable
          contains { DesignInstance versionable }
          contains { Component versionable }
          is generalization of ConfigurationDefinition
            {{ strict inherited UserField }}
```

Figure 25: ViewStates for HDL and Layout Views

There are two types of ViewStates. The first one is related to HDLViews and LayoutViews, and is defined in Figure 25. The concrete design data are stored as a bitstring in a file, whose internal structure is not known at the data model, and are handled only by the design tools. These ViewStates can also have a structural flavor,

INSTITUTO DE INFORMÁTICA
BIBLIOTECA

making references to other design objects, but the exact interconnections between these structural sub-objects are not handled by the data model.

Although there is no apparent difference between ViewStates for HDLViews and for LayoutViews, these View types are kept separate in the data model because of the strong semantic difference between them. The semantics will be expressed by particular attributes, specially those related to layout aspects, such as implementation layers and technology rules.

The second type of ViewState, defined in Figure 26, is the MHDViewState, which is related to the MHDViews. In this case, the ViewState is described in a purely structural way. There is no file where design data is stored. The data model handles not only sub-objects that make reference to other design objects, but also the exact interconnections between these sub-objects.

```
MHDViewState has Name : string versionable
      has CreationDate : time versionable
      has { Port versionable }
      has { UserField versionable }
      has { reference PredMHDViewState : MHDViewState }
      has { reference SuccMHDViewState : MHDViewState }
      contains { DesignInstance versionable }
      contains { Component versionable }
      contains { Net versionable }
      is generalization of ConfigurationDefinition
        {{ strict inherited UserField }}
```

Figure 26: ViewStates for MHD Views

To ViewStates of either type can be associated ConfigurationDefinitions. These are basically selections of objects for the sub-objects inside a ViewState. They are explained in detail later on. A ConfigurationDefinition inherits, in strict mode, all UserFields of the ViewState with which it is associated.

3.8 DesignInstances and Components

Sub-objects contained in the ViewStates are called DesignInstances. The DesignInstances may be instances of Components, which are in turn design templates that are locally declared inside the ViewStates. This declaration defines only the Component name and interface (Ports and their attributes). The specification of DesignInstances and Components is shown in Figures 27 and 28, respectively.

Components can be defined when there are many DesignInstances of the same type inside the ViewState. If there is a single DesignInstance of a given type, the user can choose to declare it directly, without the help of a Component definition.

```
DesignInstance has Name : string
               has { Port }
               has { UserField }
               [ has DesignObjectRef ]
               [ has PortMapping ]
               [ has ParameterMapping ]
               is versionable
```

Figure 27: DesignInstances

```
Component has Name : string
          has { Port }_1^n
          has { UserField }
          [ has DesignObjectRef ]
          [ has PortMapping ]
          [ has ParameterMapping ]
          is generalization of DesignInstance
            {{ strict inherited Port ; DesignObjectRef;
               PortMapping; ParameterMapping; UserField }}
          is versionable
```

Figure 28: Components

The DesignInstances must be related to other design objects through a configuration. Configurations in STAR can be expressed in many ways. A Component can be bound to another design object. In this case, all of its DesignInstances are also bound to this object. DesignInstances of the same Component can be however bound to different design objects (the two configuration options are mutually exclusive). As an example, imagine a system containing two DesignInstances MICRO1 and MICRO2, both instances of a Component Microprocessor. MICRO1 can be bound to a design object M-8080, while MICRO2 is bound to another design object Z-80.

It is also possible, however, to let a DesignInstance totally unbound inside the ViewState. In this case, the configuration is described by another object of the data model (the ConfigurationDefinition object).

Configurations are described by three elements: a reference to a design object, a port mapping, and a parameter mapping. Object references and port mappings are specified in Figure 29. The reference can be done to a specific ViewState of a given Design, but it can also be done to any other node of the control structure of this Design (to the Design, a ViewGroup, or a View). In this case, the rest of the reference, down to a specific ViewState, must be done elsewhere (either during the execution of some design tool or through a ConfigurationDefinition object).

The port mapping relates Ports of a given Component to the Ports of the design object used in the configuration of this Component (or in the configuration of each DesignInstance of this Component).

The parameter mapping, that is needed in the case that the DesignInstance or Component is bound to a parameterized object, is explained in the next section.

```
DesignObjectRef is { reference DesignRef : Design /
                     reference ViewGroupRef : ViewGroup /
                     reference ViewRef : View /
                     reference ViewStateRef : ViewState }
PortMapping has { reference DesignObjectRef . Port ;
                  reference Port }[1]
```

Figure 29: Object references and port mapping

In the specification of PortMapping, it must be noted that "reference Port" means reference to a Port of the Component where the attribute PortMapping is defined.

In the above specification, it must also be noted that the values of attributes inherited by DesignInstances from Components can be assigned either in the Component or in the DesignInstances themselves, according to the semantic of the Plasma language, so that a configuration specified in the Component is inherited by all its DesignInstances.

3.9 Parameters and parameter mapping

Parameters, that can be attached to design objects at the Design, ViewGroup, and View levels, cannot be complex elements. They have only a name and a data type, as shown in Figure 30.

```
Parameter has Name : string
              has ParameterType : { BasicType / UserType }
```

Figure 30: Parameters

The current specification of the STAR data model does not accept parametrization of the structure of a design object, where the number of DesignInstances and/or the bit widths of Ports and Nets inside a design object are defined by the value of a Parameter. This facility will be included in a next version of the data model.

The parameter mapping, specified in Figure 31, defines the values of the parameters in the case that they exist in the objects bound to the Components or DesignInstances. A DesignInstance cannot be created without an assignment of values to all Parameters of the object to which it is bound. If the configuration, however, is entirely left to a later stage, through a ConfigurationDefinition object, this assignment will also be postponed.

Values assigned to a Parameter in a DesignInstance can be constant or variable. A variable value can be the value of an attribute of the object to which the DesignInstance belongs. Both constant and variable values must match the data type of the Parameter.

```
ParameterMapping has
        { reference DesignObjectRef . UserField ; ParameterValue }_1^n
ParameterValue is { constant / reference UserField . Value }
```

Figure 31: Parameter mapping

In the above specification, it must be noted that "reference UserField . Value" in ParameterValue means the value of a UserField in the object where ParameterMapping is defined (it is defined inside Component, which is in turn defined inside either a ViewState or a MIIDViewState).

3.10 Ports

Ports are the physical interface signals through which design objects are interconnected. Ports can be single wires (see Figure 32) or bundles of wires (see Figure 33). In this latter case, the wires form a vector, whose leftmost and rightmost elements must be identified by natural numbers.

Ports can have one of three possible directions: in, out, or bidirectional (normally associated with busses). At the layout level, the Port direction has normally no meaning, although it may remain defined because of the inheritance from the control structure above the Layout View.

A data type can be associated with each Port. This data type can be either a basic data type (integer or bit-vector) or a user-defined data type.

```
Port is { PortBundle / PortWire }
PortWire has Name : string
        [ has PortType : UserType ]
        [ has PortDirection : { in / out / inout } ]
        has { UserField }
        is versionable
```

Figure 32: Port wires

```
PortBundle has Name : string
            [ has PortType : { integer / bit-vector / UserType } ]
            [ has PortDirection : { in / out / inout } ]
            [ has BitWidth ]
            has { UserField }
            contains { PortWire }_1^n
            is versionable
BitWidth is { LeftBit : integer; RightBit : integer }
```

Figure 33: Port bundles

3.11 Nets

Nets, whose specification is found in Figure 34, are the objects that model the interconnections between Ports of the interface of a design object and Ports of the DesignInstances contained in this object. A Net can connect any number of Ports. It may connect only Ports of the DesignInstances, as well as only Ports of the design object, so that direct connections between two Ports of the interface of a design object are possible.

A Net can be a single wire or a bundle of wires. In the former case, it can connect both Ports that are single wires themselves and wires of Ports that are bundles of wires. In the latter case, it can connect all wires of a Port or a subset of the wires of a Port. This subset can contain any number of either isolated wires of the Port or bundles of wires.

```
Net has Name : string
    [ has BitWidth ]
    has { ConnectedPort }_1^n
    has { UserField }
    is versionable
ConnectedPort has { reference DesignInstance . Port /
                    reference Port }
                [ has ConnectedWires ]
                has { UserField }
                is versionable
ConnectedWires has { ConnectedWireSets }_1^n
ConnectedWireSets is { ConnectedLeftBit : integer;
                       ConnectedRightBit : integer }
```

Figure 34: Nets

Ports of the object interface as well as Ports of the DesignInstances inside the object can be left non-connected. Constraints related to data types, bit width, and direction of Ports connected by a Net are not checked by the database system. They are supposed to be application-specific.

3.12 Auxiliary objects

Auxiliary Objects are objects used to support specific applications, like simulation, synthesis or test generation. Auxiliary Objects are objects other than the circuits themselves, that are created during these various applications. Examples are simulation stimuli, simulation results, test cases, testability measures, synthesis restrictions, and so on.

An Auxiliary Object has a Type (such as simulation stimuli). Types can be user- or methodology-defined. From the data model viewpoint, Auxiliary Objects are handled as bit-strings, whose internal structures are only known by the application tools. Figure 35 shows the definition of Auxiliary Objects.

AuxiliaryObject has Name : string
has Type : string
has Contents : file
is versionable

Figure 35: Auxiliary Objects

3.13 Correlations

A Correlation allows the specification of relationships between objects, according to user- or methodology-defined criteria. The Correlation criterion can be described by a special attribute which has only documentational purposes (its value is a string). A Correlation involves two objects – left object and right object – and a relationship, as shown in Figure 36.

Correlations can have directed relationships (from the left object to the right object), non-directed relationships, or bidirectional relationships.

An additional, optional attribute of the relationship – the Correlation Mode – establishes its semantics. If the Mode is "protect", then the left object of a directed relationship cannot be removed, while the left and right objects of a bidirectional relationship cannot be removed. If the Mode is "remove", then the removal of a left object of a directed relationship implies the immediate removal of the right object, while the removal of any object in a bidirectional relationship implies the immediate removal of the object at the other side. If the Correlation Mode is not specified, no special integrity constraint is verified by the database system.

The CorrelationMode has no meaning in the case of non-directed relationships. In this case, removal of an object appearing in the Correlation is possible and has no impact in the other object.

```
Correlation [ has CorrelationCriterion : string ]
            has { UserField }
            has LeftObject : CorrelatedObjectRef
            has RightObject : CorrelatedObjectRef
            has CorrelationDirection
            [ has CorrelationMode ]
            is versionable
CorrelatedObjectRef is {
        reference LibraryRef : Library /
        reference ProcessRef : Process /
        reference DesignRef : Design /
        reference ViewGroupRef : ViewGroup /
        reference ViewRef : View /
        reference ViewStateRef : ViewState /
        reference PortRef : Port /
        reference DesignInstanceRef : DesignInstance /
        reference NetRef : Net /
        reference AuxObjectRef : AuxiliaryObject /
        reference CorrelationRef : Correlation }
CorrelationDirection is {
        non-directed / directed / bidirectional }
CorrelationMode is { protect / remove }
```

Figure 36: Correlations

A typical example of use for the Correlation is the equivalence relationship. Equivalence can be established following the automatic synthesis of a design object X from another object Y. In this case, one could establish a Correlation with a directed relationship from Y (left object) to X (right object), with "protect mode" (Y cannot be removed, because it is the "father" of X, but X can be removed without affecting Y).

An equivalence can also be established following the activation of some formal verification tool. In this case, one could create a Correlation with a non-directed relationship between the two equivalent objects.

As another example, suppose that a simulation environment handles Auxiliary Objects of types SimulationStimuli and SimulationResult. An Auxiliary Object of type SimulationStimuli is applied to a Design object during a simulation session, thus creating an Auxiliary Object of type SimulationResult. In this situation, two Correlations would be needed. The first Correlation has a non-directed relationship, where the left object is the design object representation to be simulated and the right object is the applied SimulationStimuli. Even if the simulated design object representation is removed, the SimulationStimuli is maintained, since it may be used in the simulation of another representation (for instance another ViewState of the same Design). In the second Correlation, the left object is the SimulationStimuli and the

right object is the SimulationResult, and it has a directed relationship with "remove" mode. If the SimulationStimuli is removed, then the correlated SimulationResult is automatically deleted.

3.14 UserFields and data types

UserFields are attributes that can be attached to objects or to other UserFields. Their specification is shown in Figure 37. Primitive UserFields (i.e. not composed by other UserFields) must have a data type. Data types can be user- or system-defined. System-defined types are *integer*, *bit-vector*, *string*, *file*, and *time*. User-defined types can be built using *records*, *arrays*, and *sets*, as well by enumeration or subsetting on other types.

```
UserField has Name : string
          has UserFieldType : { BasicType / UserType }
          has Value
UserType is { set of BasicType / array of BasicType /
              record of BasicType / enumeration /
              subsetting / constant }
BasicType is { string / integer / file / bit-vector / time }
```

Figure 37: UserFields

4 Configurations

Section 3.8 introduced *Components* and *DesignInstances*. A DesignInstance is a sub-object in a structural description of a ViewState. It may be an occurrence of an object Component, locally defined inside this same ViewState, or it may be defined without referring to a Component. Before the ViewState can be processed by any design tool (a simulator, for instance), a design object must be selected for each DesignInstance. This selection is called a *configuration* process. This process must also include a mapping from the Ports of the selected object to the Ports of the DesignInstance and an assignment of values to the Parameters of the selected object (if Parameters do exist).

The selection of a design object can alternatively be done for the Components. In this case, all DesignInstances for a given Component are related to the same design object, and the Port mapping relates Ports of the design object with Ports of the Component. This mapping is followed by all DesignInstances of the Component. The Parameter value assignment also holds for all DesignInstances of the Component.

Choosing a complete configuration for a ViewState object X implies selecting objects $Y_1,...,Y_n$ for the DesignInstances or Components defined in X. Since each Y_i is defined by a hierarchical control structure (Design, ViewGroup, View, ViewState), where at each level several alternatives may exist, specifying a configuration means selecting, for each Y_i, one of the alternatives for each level of its control structure. The same must be done for each sub-object contained in the particular ViewState selected for Y_i, and so successively, until ViewStates without a further decomposition are selected.

Configurations may be *static*, *dynamic*, or *open* [9]. Choosing for each Y_i a ViewState without sub-objects determines a static configuration, already complete. A dynamic configuration is obtained when a ViewState with sub-objects is selected, or if only a Design, or one of its ViewGroups or Views, is selected. In this case, there are still many possible representations for the sub-objects Y_i and thus many possible configurations for X. The dynamic configuration of X must be completed later on, by making choices so that, for each Y_i or sub-object contained within them, ViewStates without sub-objects are selected. An open configuration exists when, within X, no choice of objects is done for the Components and/or DesignInstances of X. The open configuration must be completely defined later on.

The selection of a design object for a Component (or alternatively for the DesignInstances) may be *direct* or *indirect*:

- in the direct selection, the objects Y_i to be bound to the Components or DesignInstances are specified inside X; this possibility supports both static and dynamic configurations;
- in the indirect selection, the specification of the objects Y_i is done (or completed) in another special object, called a *ConfigurationDefinition*; this possibility supports both dynamic and open configurations.

The ConfigurationDefinition object is thus used for completing a dynamic configuration as well as for completely resolving an open configuration. It is associated with the ViewState X to which the configuration belongs (see Figures 25 and 26). A ViewState may have several ConfigurationDefinitions associated with it. They may be complementary to each other, that is, each specifies the configuration for one or several Components or DesignInstances of the ViewState. A single ConfigurationDefinition may however specify the configuration for all Components or DesignInstances.

If a ViewState X already binds objects to its Components or DesignInstances, but without completely specifying the configuration, as in the dynamic configurations, the ConfigurationDefinition objects CD_1^X,...,CD_p^X that are associated with X must complete the configuration. For each Component or DesignInstance of X, the initial object in CD_j^X may be a Design, a ViewGroup, a View, or a ViewState, but the final object must be a ViewState. If, for instance, a DesignInstance DI_i of X already refers to $D_2.VG_1$ (a ViewGroup VG_1 of a Design D_2), the ConfigurationDefinitions associated with X must designate the alternatives chosen at each level of the control structure of D_2 below VG_1, until one of its ViewStates.

Configurations may be nested. Therefore, a configuration may refer to other configurations to complement a description. Suppose that, for the already considered DI_i of X, a ConfigurationDefinition CD_j^X has selected a View V_3 and its ViewState VS_2 for completing the reference $D_2.VG_1$. Let us call Y the complete reference to this ViewState. If Y has sub-objects, and ConfigurationDefinitions CD_1^Y,...,CD_m^Y had been already created for it, then CD_j^X may bind a ConfigurationDefinition CD_k^Y to DI_i.

Figure 38 shows the specification of the ConfigurationDefinition objects. A ConfigurationDefinition inherits all UserFields of the ViewState to which it belongs and may have additional UserFields.

```
ConfigurationDefinition has Name
                        has { ConfigItem }_1^n
                        has { UserField }
ConfigItem is { ComponentConfig / InstanceConfig }
ComponentConfig has reference CompRef : Component
                    has ObjectRef
                    has PortMapping
                    has ParameterMapping
InstanceConfig has reference DesInstRef : DesignInstance
                   has ObjectRef
                   has PortMapping
                   has ParameterMapping
ObjectRef is { reference ViewStateRef : ViewState /
               reference ConfigDefRef : ConfigurationDefinition }
```

Figure 38: ConfigurationDefinitions

A configuration manager for the STAR framework is under specification [10]. It will cover the following aspects:

- to propose and implement a specialized language for the specification of configurations according to design constraints, such as expressions involving attribute values;
- to propose and implement a configuration tool that allows the specification of configurations through either this specialized language or an interactive dialogue under user control.

Acknowledgments

The STAR data model is strongly based on the GARDEN data model, specified at the IBM Rio Scientific Center by a group of researchers. Special thanks are due to Arnaldo H. Viegas de Lima, who has a leading role in the GARDEN project and gave a valuable contribution to the STAR data model specification.

A special acknowledgment deserves also Lia G. Golendziner, who made a careful analysis of the Plasma language and gave valuable hints for a much more clear notation and for obtaining better consistency with current data modelling concepts.

All aspects of the data model that are related to the configurations have been defined jointly with Helena Grazziotin, who is responsible for the specification and implementation of the STAR configuration manager.

ATICA
BIBLIOTECA

References

[1] D.S. Harrison et al. Data management and graphics editing in the Berkeley Design Environment. In *International Conference on Computer Aided Design*, IEEE, 1986.

[2] W.D. Smith et al. FACE Core Environment: the model and its application in CAE/CAD tool development. In *26th Design Automation Conference*, ACM/IEEE, 1989.

[3] J. Daniell and S.W. Director. An object-oriented approach to CAD tool control within a design framework. In *26th Design Automation Conference*, ACM/IEEE, 1989.

[4] K. Gottheil et al. The CADLAB workstation CWS - an open, generic system for tool integration. In F.J. Rammig, editor, *IFIP Workshop on Tool Integration and Design Environments*, North-Holland, 1988.

[5] F.R. Wagner, A.H. Viegas de Lima, L.G. Golendziner, and C. Iochpe. STAR: um ambiente para a integração de ferramentas de projeto de sistemas digitais. In *VI Congresso da Sociedade Brasileira de Microeletrônica*, SBMICRO, Belo Horizonte, 1991.

[6] F.R. Wagner, C. Freitas, and L.G. Golendziner. The AMPLO system - an integrated environment for digital systems design. In F.J. Rammig, editor, *IFIP Workshop on Tool Integration and Design Environments*, North-Holland, 1988.

[7] E.B. de la Quintana, G.O. Annarumma, and P. Molinari Neto. *GARDEN - the Design Data Interface.* Technical Report CCR-107, IBM Rio Scientific Center, Rio de Janeiro, 1990.

[8] F.R. Wagner and A.H. Viegas de Lima. Design version management in the GARDEN framework. In *28th Design Automation Conference*, ACM/IEEE, 1991.

[9] F.R. Wagner. *Modelos de Representação e Gerência de Dados em Ambientes de Projeto de Sistemas Digitais.* Technical Report CCR-121, IBM Rio Scientific Center, Rio de Janeiro, 1991.

[10] H. Grazziotin. *Gerência de Configurações no Ambiente STAR.* Master's thesis, CPGCC / UFRGS, 1991. (under development).

Relatórios de Pesquisa

RP-167: "The Data Model of the STAR Framework", novembro 1991.
F.R. WAGNER

RP-166: "Design Methodology Management in Design", novembro 1991.
F.R. WAGNER

RP-165: "Desenvolvimento de "Asa" e "Trama"", outubro, 1991.
S.D. OLABARRIAGA; E.M. CORRÊA; C. CALLIARI; A.L. POMPERMAYER

RP-164: "Estudo Topológico e Elétrico da Nova Célula de Base para CIs Gate Array - Tecnologia 1.2 um", outubro, 1991.
L.R. FROSI; G.V. PAIXÃO; J.L.G. CUNHA; D.A.C. BARONE

RP-163: "Representação de Conhecimento em Engenharia do Conhecimento", setembro, 1991.
N. EDELWEISS

RP-162: "Biblioteca de PADS Digitais CMOS 1.5 um - Versão 1", setembro 1991.
M.K. DOSSA

RP-161: "Versões Intervalares do Método de Newton", agosto 1991.
H. KORZENOWSKI; M. LEYSER; T.A. DIVERIO; D.M. CLAUDIO

RP-160: "Processamento Vetorial e Vetorização de Algoritmos na Máquina Convex C210", agosto 1991.
T.A. DIVERIO

RP-159: "Um estudo de técnicas de validação e de verificação de produtos de software", junho 1991.
N. EDELWEISS

RP-158: "Extensão das Ferramentas PIU/LINUS de especificação e controle de interfaces com o usuário", Junho 1991.
J.P. FIGUEIRÓ

RP-157: "The Domain of Nets and the Semantic Bases of a Notation for Nets", Abril 1991.
A.C.R. COSTA

RP-156: "Continuous Predicates and Logical Reflexivity", Abril 1991.
A.C.R. COSTA

RP-155: "TENTOS - Gerenciador de Software para Microeletrônica", abril 1991.
F.G. MORAES; R.A.L. REIS.

RP-154: "Sistemas Especialistas para a Engenharia de Software", Abril 1991.
H. AHLERT.

RP-153: "Estudo Comparativo e Taxonomia de Ferramentas de Suporte à Construção de Sistemas, Abril, 1991.
H. AHLERT.

RP-152: "IMP-MAC - Emulador de Impressora Padrão Apple", Abril, 1991.
C. DE ROSE; R.F. WEBER.

RP-151: "Em direção a um modelo para representação de aplicações de escritórios baseadas em documentos", Abril, 1991.
D.B.A. RUIZ.

RP-150: "Implementação de Sistemas de Gerência de Banco de Dados", Abril, 1991.
D.B.A. RUIZ.

RP-149: "Integração de Ferramentas no Sistema AMPLO: Crítica e Proposta de Extensões", março, 1991.
F.R. WAGNER.

RP-148: "Interface de Entrada para Teclado e Mouse", março, 1991.
J.M. DE SÁ.

RP-147: "Servidores - Guia do Usuário - Edição 1", janeiro, 1991.
A.R. TREVISAN, C. LEYEN, G. CAVALHEIRO, J.F.L. SCHRAMM, L.G. FERNANDES, P. FERNANDES, R.M. BARRETO, R. TEODOROWITSCH

RP-146: "Biblioteca de Células TRANCA regras ECP15/1", janeiro, 1991.
C. CRUSIUS, L. FICHMAN, M. KINDEL, C. MARCON, R. REIS

RP-145: "Manual do Usuário do Projeto TRANCA; v 1.0", janeiro 1991.
F.G. MORAES; M. LUBASZEWSKI, R.A.L. REIS

RP-144: "Manual do Sistema TRAMO Projeto TRANCA versão 1.0", janeiro 1991.
M.A. SOTILLE; C.E.S. SOUZA; M.G.R. ARAUJO; M. LUBASZEWSKI; R.A.L. REIS

RP-143: "PILCHA: Projeto e Implementação de um Sistema Digital Discreto dedicado ao controle de acesso direto a memória", janeiro 1991.
F. AZEREDO; L. ROISENBERG; D.A.C. BARONE

RP-142: "Ambiente para Estudo de Fractais - Relatório de Projeto", janeiro 1991.
S.D. OLABARRIAGA; F.S. MONTENEGRO