Design Management Requirements for Hardware Description Languages

Flávio R. Wagner
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
e-mail: flavio@inf.ufrgs.br

Abstract
This paper discusses the relationships between HDL’s and design management services commonly found in EDA environments and frameworks. By using VHDL as a case study, it is demonstrated that HDL’s are not neutral with regard to these services. The paper shows that the flexibility in using a language together with the widest possible range of design management mechanisms should be a major language requirement. The paper indicates VHDL extensions and modifications that follow in this direction.

1 Introduction
Integrity constraints, in the design of complex electronic circuits and systems, may be classified into three different groups: fine-grain data constraints, which are related to a data representation model, coarse-grain data constraints, mainly related to a data management model, and methodology constraints, related to the goals that are expected in a particular design process. Design environments and frameworks usually verify these three classes of constraints by different mechanisms or services. While fine-grain data constraints are normally verified by the design tools themselves, coarse-grain data constraints and methodology constraints are verified by domain-neutral tools that offer application-independent services.

Although the corresponding constraints are usually verified by separate tools or mechanisms, these three aspects of the design – data representation, data management, and design methodology management – are not in fact completely independent from each other. A tight coupling between tools and services would allow a more flexible and powerful control of the design process, by means of an integrated and complementary verification of constraints [1].

A hardware description language implements a given data representation model which is suitable for a given range of design abstraction levels. This model defines a set of fine-grain data constraints that can be verified through the language, either static constraints, to be verified by the compiler, or dynamic constraints, usually verified by a simulator.

However, very often HDL’s also impact the other aspects of the design process control. Certain language constructs imply design management models and define a set of related integrity constraints. The consequence for an environment based on such a language is a restriction on the possibilities it has to implement a flexible design process control.

A thesis can be thus formulated from the above rationale: an HDL should not unnecessarily restrict the set of design management models that a design environment based on such HDL could possibly implement. The language should be as flexible as possible with regard to design management aspects, in order to allow its use together with a variety of framework management services.

This paper analyzes VHDL according to the above thesis and shows that it indeed restricts the space of design management models a VHDL-based environment can implement. Based on this analysis, the paper gives indications on which language modifications should be considered to allow VHDL to be more consistently used in conjunction with design management mechanisms that have been proposed in recent years in the context of EDA frameworks.

The remaining of this paper is organized as follows. Section 2 discusses design integrity types in the context of EDA frameworks. Section 3 deals with the special problem of managing the various design object representations created along the design process. From the previous discussion, a set of language requirements is derived in Section 4. VHDL is thus analyzed according to these requirements in Section 5, which also suggests language modifications. Section 6 draws conclusions and proposes further work.

2 Design Integrity
In this paper, integrity constraints are considered in a very general sense, including all those aspects that must be verified during a design process to allow the designer to achieve a circuit or system which exhibits the desired behavior and fulfills requirements such as area, power, fabrication cost, design time, design cost, etc. In this broader perspective, realizing a given function under certain input conditions is also a constraint to be verified. Different constraints are handled by different tools running on a design environment. These tools include both domain-specific design tools, such as simulators, and domain-neutral tools, that implement application-independent services like version and
methodology management. Three classes of design integrity constraints may be thus identified: fine-grain data constraints, coarse-grain data constraints, and methodology constraints.

Fine-grain data constraints are related to a data representation model and are normally handled through the design tools themselves. They correspond to both static and dynamic design aspects. Static constraints, like the data type compatibility of interconnected signals and the adherence of a layout to a set of technology rules, can be verified by language compilers and other static verifiers. Dynamic constraints are related to the correct behavior of the design objects and are verified by tools like simulators and formal provers. Examples are the fulfillment of setup and hold times and the correct implementation of an arithmetic operation by an ALU.

Coarse-grain data constraints are derived from relationships between separate objects or different representations of the same object. Examples are:

1. components of a given configuration must share common attributes (e.g. they must have a power consumption less than a given value);
2. different versions of the same object must share common attributes (e.g. they must share certain interface signals); and
3. an equivalence relationship can be established only between versions of the same object.

In an EDA framework, these constraints are usually verified by domain-neutral, data management tools, not by design tools. Since they are of different natures, however, they may be handled by different tools. The first constraint above is clearly dependent on a given design goal. A configuration manager should allow the user to set up these constraints for each design. The second constraint, in turn, could be "hardwired" into a version management tool, since it is always expected to hold. In the case of the second constraint it seems preferable to give the user (or the project manager) freedom to decide which attributes must be shared by the different versions of the same object.

Methodology constraints, finally, are related to the specific goals established for a particular project. They are usually derived from design requirements, such as area, power consumption, operating frequency, fabrication and design costs, etc., and should never be enforced by design tools. They are usually verified by domain-neutral, design methodology management tools (also called task flow management tools or design process management tools).

The main conclusion from the above discussion is that a design environment, through its various instances, should adequately cover all classes of constraints, without interferences and omissions. If a given constraint type is already verified by a given tool (either a design tool or a domain-neutral tool), then no other tool should verify the same constraint. If a domain-neutral tool allows the designer or project manager to define design-specific constraints in a flexible way, then these constraints should not be "hardwired" into the design tools, since that would make that framework service innocuous. Therefore, this flexibility must be reflected by the design tools.

These facts impose a very strong requirement on the design of both design tools and framework services. Framework developers are considering powerful management services that should not remain unnoticed by tool developers. However, if there are no standards for the framework services (and one can argue whether these standards are really possible or desirable to define), then tool developers face a serious problem. Different frameworks will implement different models for the same design management services, or even offer different services. The solution seems to be the development of design tools that are as flexible as possible with regard to the verification of coarse-grain data constraints and methodology constraints, so as to be compatible with a variety of design management mechanisms and policies.

3 Design object representations

During the design process, for each object a multitude of representations is created, resulting from the design evolution along three main dimensions. Various views are created, corresponding to descriptions at different abstraction levels (algorithmic, RT, logic, layout, etc.). Various alternative architectures or implementations are analyzed, according to different trade-offs between speed, area, power consumption, etc. Finally, for each design alternative, many consecutive improvements (revisions) are created at each abstraction level. A design environment must support the evolution along these three dimensions in an independent, yet related way.

Consider the following design methodology. After scheduling and allocation, an initial algorithmic description is transformed into an abstract structural RT description, specified in terms of generic components (registers, adders, etc.). Since scheduling has been already performed, this description defines the exact timing of register transfers in terms of clock cycles. The clock, which did not exist in the original algorithmic specification, must from now on exist as an interface signal in all object representations to be created at lower abstraction levels. However, the clock in this abstract RT description is also an abstract entity. When components are selected in a later technology mapping step, register types for a particular technology are defined. As a consequence, and assuming dynamic registers are chosen, the abstract clock is specialized into either a 2- or a 4-phase clock, according to the alternative taken by the designer.

The example shows two modeling principles which are likely to occur in the design process and that relate different design representations: inheritance and specialization. The basic interface signals (data and address lines, for instance, as well as control signals that already exist in the initial specification) already appear at the algorithmic description and must be inherited by all other representations of this object. At the abstract structural RT level, a clock is added to this basic interface. At the concrete structural RT level, the clock is specialized into either a 2- or a 4-phase clock.
The example illustrates that views and alternatives may both inherit interface definitions and specialize them, by either adding new signals or refining signal definitions. There is no single pattern of behavior for the possible relationships between views and alternatives, since these relationships are extremely application-dependent. However, most environments implement only a generic versioning mechanism, where no distinction or relationships between views, alternatives, and revisions can be handled. Other environments do support a distinction, but enforce a fixed scheme for expressing the relationships that exist between these dimensions.

Flexibility should be provided, at least, in the definition of the following aspects of management: (a) the set of views and alternatives to be created for each design object; (b) how views and alternatives are chained together along the design evolution; (c) the attributes (typically interface definitions) to be defined for each view and alternative; and (d) the inheritance and specialization of attributes that is mandatory and for each new design representation.

This section has shown that, although handling of versions (views, alternatives, and revisions) is in most environments considered a pure data management issue, it has in fact many relationships to data representation aspects. This coupling between data management and representation, which is expressed by a set of methodology-specific integrity constraints in form of inheritance and specialization relationships, is thus very helpful to a semantically meaningful management of design methodologies. Furthermore, since these constraints are methodology-specific, they should be defined in a fixed way neither by design tools nor by framework tools.

4 Language requirements

Language-based tools, such as compilers, simulators and synthesis programs, are among the most widely used design tools. Requirements for the development of design tools, concerning the verification of the various kinds of integrity constraints, and in particular the handling of the relationships between design representations, must then also apply to language-based tools and hence influence the definition of the languages.

The main purpose of language-based tools is the verification of fine-grain data constraints. These constraints are derived from a data representation model which is implemented by the language, encompassing a given range of design abstraction levels and application domains. Restrictive language features should be added only if directly derived from this representation model. Features that are related to management aspects should be carefully studied, so as to guarantee that they maintain the greatest possible flexibility and do not interfere with possible framework management services. Coarse-grain data constraints should never be defined in a permanent way as a built-in language feature. The management mechanisms that are implemented by the language should be very basic and flexible ones, so that each application could define particular management policies that enforce the desired problem-specific constraints.

However, if we analyze currently available languages, we recognize that they implement models that are restricted not only with regard to representation but also to management aspects. This analysis, regarding VHDL, will be presented in next section.

There is an obvious feedback between languages and design environments. Integrity constraints that are permanently established in the language definition exist in any design environment based on this language. Integrity constraints to be defined by the user or by the design methodology, in turn, could be expressed through either the language or framework services, but in any case there must be a consistent solution. If a constraint is already expressed through the language, it must be recognized, accepted, and enforced by the framework tools. Eventually, the framework could provide mechanisms for overriding constraints expressed through the language. If a framework service gives the user flexibility in the definition of certain types of constraints, this flexibility should not be made innocuous by a built-in, permanent language feature.

5 VHDL analysis

In the following, VHDL features are analyzed with regard to the requirements that have been developed in the previous sections. Restrictions that should not be imposed by the language are identified, and language extensions or modifications to avoid those restrictions are sketched. Extensions are also proposed to allow the language to give adequate support to commonly available framework services. In order to do that, data representation issues are separated from design management ones. The analysis concentrates on some important language features that build a kind of VHDL design management model: the management of design object representations, the inheritance and specialization of attributes, the management of configurations, and the management of libraries.

5.1 Alternatives, views, and revisions

VHDL supports a very simple model for the management of the various representations of a design object, namely that an entity may have several architectures. According to the classification introduced in Section 3, the architecture concept is thus a generalization of alternatives, views, and revisions. Since there is no specialization supporting a distinction among these types of representations, a VHDL user cannot know which type was meant for a particular architecture of a given entity. Many design environments, however, do support such distinction.

In theOct manager [2], for instance, a cell has many views, each view has many facets, and each facet has many versions. Facets describe the interface and contents of the cells and model aspects of a cell that are visible to particular design tools. The Oct views and facets correspond to different specializations of views. Oct versions correspond to revisions. Alternatives are not supported.

In the Damascus system [3], in turn, a design object has many representations (corresponding to views),
each representation has many alternatives, each alternative has many revisions, and each revision has many design stages. The Damascus revisions and design stages are different specializations of revisions.

The STAR framework [4], finally, organizes the representations of a design object as a hierarchy of ViewGroups, with any desirable depth. ViewGroups gather other ViewGroups and Views. ViewGroups have a close resemblance to alternatives, but they allow a more powerful organization of the representations, in a way that is already oriented to design methodology management [1]. Each View may have several View-States, corresponding to revisions.

A VHDL-based environment is not capable of emulating completely any of the above three models for the management of the design representations. The distinction among views and alternatives, as in STAR and Damascus, the specialization of views, as in Oct, and the hierarchical organization of alternatives, as in STAR, remain impossible. However, a possible service external to VHDL could handle various revisions for a single architecture, as well as specializations of revisions, as in Damascus. The VHDL analyzer should be aware of this service, to be able to select revisions for the referenced architectures.

It seems desirable to support at least the distinction among views and alternatives. This could be implemented in VHDL by introducing a third concept, for instance views, besides entities and architectures. An architecture would then correspond to alternatives and would gather any number of views. It remains an open question whether to support the specialization of views and the hierarchical organization of alternatives or not. These two mechanisms would be naturally supported by an object-oriented approach, where an entity would correspond to a class that could be specialized into sub-classes in a hierarchical way.

5.2 Inheritance and specialization of attributes

Although the definition of interface aspects and other object attributes is clearly a data representation issue, their association to object representations and their inheritance among representations are data management issues. Parameters (like VHDL generics) model object attributes, but their very existence is already a data management issue, since they are intended to support the declaration of generic, reusable object definitions.

In VHDL, interface ports, generics, and user-defined attributes are inherited by all architectures of the same entity. There is no mechanism allowing the specialization of either ports, generics, or user-defined attributes in architectures. However, distinct and sometimes more powerful inheritance and specialization mechanisms exist in many design environments, such as the already mentioned Oct, Damascus, and STAR systems.

In the Damascus system, for instance, the object interface is defined in the alternatives, not in the design objects or in the representations, in a modeling approach which is the opposite of VHDL, which defines the interface in the entities, not in the architectures.

Damascus revisions inherit the interface defined in the corresponding alternative and can add new interface attributes, thus specializing the interface definition.

In a similar approach, in the Oct manager the object interface is defined in the contents facet, which is a special facet of a view, not in the cells or in the views. This basic interface is inherited by the other interface facets of the same view. Furthermore, interface facets can add additional attributes to a view, that are not necessarily related to the interface ports, but more generally to any aspects that are externally visible by particular tools, as routing channels, for instance.

The STAR framework, finally, adopts a more general model, which is a combination of the ideas found in VHDL, Oct, and Damascus. Interface aspects can be defined at any level of the representation hierarchy (the so-called object schema): at the root (corresponding to a VHDL entity), at the intermediate ViewGroups (corresponding to alternatives), and at the leaves (the Views). Each node of the hierarchy inherits interface aspects from the ascending nodes and can both add new aspects and specialize already defined aspects. Furthermore, not only interface aspects can be inherited, but also parameters and any other attributes not necessarily related to the interface (as in the Oct manager).

The STAR framework implements an instance inheritance mechanism [5]. This means that both the attribute existence and the attribute value (when defined) are inherited. Furthermore, two types of inheritance are supported. In the default inheritance, an attribute definition or value may be changed in a descending node of the object schema. In the strict inheritance, an attribute definition or value cannot be changed. It is up to the user to specify which type of inheritance is attached to a given attribute.

It seems desirable to take a more flexible approach in VHDL, allowing the definition of interface aspects and generics at both the entities and architectures. If this approach is combined with the introduction of the view concept, as already proposed in the previous subsection, and views, as architectures, may also add or specialize interface aspects and parameters, a solution which is similar to the one taken in the STAR framework arises.

VHDL already supports the association of user-defined, general purpose attributes to entities and architectures, as well as the inheritance of attributes from entities to architectures. However, attributes must have constant values, both the attribute existence and value are inherited, and the value cannot be changed in the architecture. The introduction of the instance inheritance and default inheritance concepts would result in a more flexible modeling approach.

5.3 Configuration management

Design objects to be handled in design environments have a complex and hierarchical structure, with objects composed of other sub-objects. A concrete representation of a composite object, where sub-objects are bound to particular versions of other objects, is called an object configuration. Although the modeling of composite objects is a data representation
issue, facilities for modeling and building configurations are a data management issue.

There are three types of configurations: static configurations [6] associate a complete reference (object and version) to each sub-object; dynamic configurations [6] associate partial, incomplete references to the sub-objects; and open configurations leave the reference completely undefined. Static and dynamic configurations must be resolved to a complete reference before the object may be used by certain design tools.

VHDL supports all three types of configurations. Static configurations are defined through configuration specifications within architectures, which allow the user to bind a given sub-object to a pair entity - architecture. Dynamic configurations are defined in a similar way, but only an entity is selected, and the architecture is chosen later on. This choice can also be expressed in VHDL, by using a separate design unit called configuration declaration. In order to implement open configurations, VHDL supports the declaration of local components within a given architecture. Components are declared through their interface and generics, and may be instantiated as sub-objects. Binding of components to other design objects is done later, through a configuration declaration.

In fact, components would be needed only for supporting open configurations, but VHDL imposes their declaration and instantiation also when the designer intends to use static or dynamic configurations. In the case of static configurations, for instance, components impose an indirect reference that could be avoided: a sub-object references a component, and this in turn is bound to a pair entity - architecture. In the STAR framework, for instance [7], a sub-object (called a DesignInstance) may instantiate another design object either indirectly through a locally declared component or directly. This flexibility could be added to VHDL. It would ease the handling of static and dynamic configurations for composite objects with few sub-objects, something that is likely to occur at high design abstraction levels.

There are two different approaches for resolving open and dynamic configurations. In a framework-based approach, the framework offers a configuration manager, a special service which supports facilities for expressing user-defined constraints that help select desired objects and/or versions. In the STAR framework, for instance, the configuration manager offers a special selection language for the description of design qualities the user expects from the versions. Although this approach is consistent with VHDL, since configurations are declared and resolved through mechanisms that are complementary and external to the language, the VHDL analyzer must be aware of these selection mechanisms, if it is expected that it remains responsible for resolving configurations.

In a language-based approach, in turn, the user-defined constraints for selecting the desired objects and/or versions are expressed through language constructs. An extension to the language is needed in this case, since VHDL has currently no means for expressing object qualities for solving open or dynamic configurations. This approach is followed in the SDE environment [8], for instance. It seems highly desirable to extend VHDL in this direction, since none of the alternatives may be of interest for certain users: to have a framework with the desired configuration management services; to use only static configurations; or to manually solve dynamic and open configurations, by searching objects through the database interface and building appropriate configuration declarations.

5.4 Workspaces

Workspaces are used in design environments for many different purposes. For access control, they serve to hide sensitive information from other designers. For cooperation management, they help organize information to be shared by members of a team, so that each member may have private access to working descriptions while sharing already released descriptions with other people. In this latter case, shared descriptions may be stored in various workspaces (commonly referred to as libraries) according to organizational criteria. For configuration management, a workspace may gather representations that can be used together in a meaningful configuration, so that dynamic configurations can be automatically solved by specifying from which workspace a version for the selected object has to be retrieved. This mechanism is found in the Version Server [6], for instance, where workspaces are called layers. For revision control, a workspace may be associated to a design status of the versions, such as in progress, stable, frozen, and released. To each design status different operations (remove, edit) may apply. A design status may also be used for configuration management purposes, as in Biliris [9], if the operation reference is added to the controlled operations, and for cooperation management purposes, as in Kiler [10], if the operation lend is added.

In many systems (for instance [2, 11]), workspaces serve simultaneously many or all of the above mentioned purposes. No matter which is the primary purpose, workspaces are normally associated with a visibility mechanism, by which users and/or tools see only design representations in selected workspaces. Additional constraints and operations may be associated to design representations in each workspace, depending on the purposes of the workspace concept.

VHDL has a very simple library concept. Design units may be stored in various libraries, whose contents are made visible to other design units through library clauses. The mapping between VHDL libraries and directories of the host file system is performed externally to the language.

As already stated in subsection 5.1, an external revision control mechanism could be easily added to VHDL. It could be associated to concepts like workspaces and revision design statuses, as mentioned above. Therefore, for the sole purpose of revision control the VHDL library concept presents no restriction.

For the remaining purposes, however, the language does not provide the necessary support. For access control, the VHDL analyzer should be aware of the privileges granted to the current user, so as to deny access to particular libraries. For cooperation management purposes, a similar access control would be
combined with the already provided organization of design units in various libraries and with some external mechanism for constraining operations (remove, edit, lend) on objects in different libraries. It must be noted that this external mechanism is not restricted by any existing language feature.

For configuration management purposes, finally, a workspace-based approach imposes that each workspace contains at most one version of each design object, so that the automatic resolution of dynamic configurations can be based on the locality of the versions. This seems not to be compatible with the workspace concept when applied to access or cooperation management purposes, where this restriction would be too hard. It remains therefore to be investigated whether a distinct workspace mechanism should be added to the already existing VHDL library mechanism, since there are alternative solutions to the configuration management problem. In the case of a positive answer, the workspace concept should be combined to the expression of dynamic configurations in configuration specifications and declarations, so that the analyzer could automatically select the architecture (or revision, if a revision control mechanism is also available) from the specified workspace.

6 Conclusions and future work

By using VHDL as a case study, this paper has shown that there is an interplay between hardware description languages and design management services. Any language implements a data model that considers not only data representation aspects, but also design management aspects. It is of course desirable that a given language can be used with the largest possible variety of design management services. For that purpose, the language should not include constructs that unnecessarily restrict the design management models and policies with which it can be used. This seems to favor the removal of design management aspects from the language. However, only the availability of certain language constructs can make possible the use of the language together with other design management services. If these constructs cannot be avoided, they should remain as flexible as possible.

The paper discussed four main management aspects that affect VHDL: the management of the design representations, the inheritance and specialization of attributes among representations, the configuration management, and the management of workspaces. In all these cases, it has been shown that VHDL presents certain restrictions that should be removed, either by extending or modifying the language. In certain cases, it has been further detected that interesting design management services could not be used together with VHDL without extensions to the language.

All the suggestions that have been sketched in this paper follow the same direction. They try to give the language the maximum flexibility, in order to be used with the widest possible range of available design management models.

Although the paper compared VHDL to management models from many different systems, many proposals follow particular models available in the STAR framework. The use of this framework as a paradigm is due to two important features it presents. Firstly, it implements very flexible management services, and flexibility has been established as a strong design management requirement for hardware description languages. Secondly, the STAR models for data representation, data management, and design methodology management are tightly coupled to each other, and this is a necessity that is not always recognized in design environments, although this fact becomes evident when one think of a design language like VHDL, where many constructs affect both data representation and design management aspects.

A natural evolution of the work presented in this paper is the implementation of a design environment using the STAR services and based on a VHDL dialect containing the modifications and extensions that have been sketched here.

References


