RAPPORT DE RECHERCHE

PREVAIL-DM: A FRAMEWORK-BASED ENVIRONMENT FOR FORMAL HARDWARE VERIFICATION

Flávio Rech Wagner

RR-899 **Juillet 1992**

Doador

FL 2348

05/03/2015

R$ 0,01

PREVAIL-DM: A FRAMEWORK-BASED ENVIRONMENT FOR FORMAL HARDWARE VERIFICATION

Flávio Rech Wagner*

Abstract

This report describes the framework-based PREVAIL-DM design environment for formal hardware verification. PREVAIL-DM integrates proof tools that are available in the PREVAIL environment around a common, VHDL-based conceptual schema. Tools are encapsulated according to a black-box approach. Design data are stored in a unique data base, and the environment offers a common main user interface, which gives access to design tools and methods and allows browsing through the database objects. Available methods to be applied on the design objects are oriented to the application semantics, thus helping to maintain all desired schema-related integrity constraints. PREVAIL-DM is implemented upon the Cadlab framework and uses most of its main features.

Resumé

Ce rapport présente l'environnement de projet PREVAIL-DM, orienté vers la preuve de circuits et systèmes électroniques et basé sur un *framework*. PREVAIL-DM intègre des outils de preuve disponibles dans l'environnement PREVAIL autour d'un schéma conceptuel de données commun et basé sur VHDL. Les outils sont encapsulés selon une approche *black-box*. Les objets de projet sont stockés dans une base de données unique, et l'environnement offre une interface-usager communne qui permet l'accès à tous les outils ainsi que la navigation à travers la base de données. Les méthodes qui sont exécutables sur les objets de projet sont orientées vers la sémantique de l'application, de façon à aider l'usager à préserver toutes les contraintes d'intégrité liées au schéma. PREVAIL-DM est réalisé en utilisant le *framework* Cadlab comme plateforme et se sert de la plupart de ses plus importants services.

* On leave from the Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil

Contents

1 INTRODUCTION

Design frameworks [1] are general-purpose platforms for building application-specific, integrated design environments. They offer services like data management mechanisms, usually comprising facilities for modelling complex electronic systems and a common database system, tool integration mechanisms, version and configuration management, design methodology management, data distribution with concurrency control, and support for building new, integrated tools, such as mechanisms for the construction of uniform user interfaces and for intertool communication.

PREVAIL [2] is an environment that offers specialized tools for the formal proof of hardware. It allows the user to specify hardware structure and behavior by means of a subset of VHDL [3], and offers translators that automatically generate proof-oriented internal representations. The environment offers three different provers (TACHE, LOVERT, and the Boyer-Moore theorem prover), that are applicable to combinational and certain classes of sequential circuits.

This report describes PREVAIL-DM, a framework-based version of PREVAIL which integrates the above mentioned proof tools around a common, VHDL-based conceptual schema. The main motivation for this development is the addition of powerful data management facilites to the PREVAIL environment, and in particular the automatic verification of integrity constraints during the design and proof processes. As a main requirement for the PREVAIL-DM development, a "black-box" tool integration approach was adopted, because changing the tool codes would be a too great effort.

PREVAIL-DM design data are stored in a unique data base, and the environment offers a common main user interface, which gives access to design tools and methods and allows browsing through the database objects. Available methods to be applied on the design objects are oriented to the application semantics, thus helping to maintain all desired schema-related integrity constraints. PREVAIL-DM is implemented upon the Cadlab framework [4] and uses most of its main features.

The conceptual schema (object classes, methods, and relationships) gives special support for the proof of equivalence and implication between VHDL architectures, as well as for the proof of properties expected from these architectures. It includes a specific, powerful support for the utilization of the Boyer-Moore prover by experienced users.

PREVAIL-DM thus supports some of the framework-supported features expected from integrated design environments, mainly those related to data management (thus the name PREVAIL-DataManagement) and consistency. Future work will specially concentrate on support for design methodology management.

The reminder of this report is organized as follows. Section 2 gives an overview of the main features of PREVAIL-DM. Section 3 details the conceptual schema. Section 4 describes implementation aspects, such as the integration of the design tools according to the black-box approach and the schema realization by means of the TIDL language offered by the Cadlab framework. A detailed analysis of the environment features can be found in Section 5. Finally, Section 6 discusses future work.

2 GENERAL OVERVIEW

Main features

PREVAIL-DM presents some of the main features expected from an integrated design environment implemented upon a design framework. The design tools are integrated around a common conceptual schema, in such a way that their execution preserve application-specific integrity constraints. The design data are stored in a unique data base, thus avoiding data redundancy and inconsistency. The tools are activated from a common, graphical-interactive user interface based on Motif, which also gives access to a database browser. The schema can be easily extended so as to include new object classes and methods.

Other useful features expected from integrated environments and / or supported by design frameworks that are not implemented in PREVAIL-DM are discussed in Section 5.6.

Cadlab framework

PREVAIL-DM is implemented upon the Cadlab framework and uses most of its main services, as introduced below. Other services supported by the framework that are not used in PREVAIL-DM are discussed in Section 5.7.

The basic layer of the Cadlab framework is a database system which implements a general-purpose data model, named IDM [5], specially conceived for representing complex design objects. The PREVAIL-DM schema is defined by means of the object-oriented modelling facilities of the TIDL language [6, 7] available in an additional framework layer. For each object class the schema defines methods, attributes, and a graphical representation. Facilities available in both layers allow the definition of the application-specific PREVAIL-DM access management policies.

PREVAIL-DM automatically inherits very useful features presented by the Cadlab framework. The framework automatically builds the main user interface of the design environment: the graphical-interactive MOTIF-based framework *dektop* [8] gives access to the application-specific methods and allows browsing through the database objects and relationships. The TIDL language supports an easy extension of the environment, allowing the definition of new object classes, relationships, and methods, as discussed in Section 5.4. The database system can be used in a distributed way over a network of UNIX-based workstations and controls the concurrency of user accesses.

Conceptual schema

Design tools are integrated around a VHDL-based, common conceptual schema. It is a coarse-grain schema, where objects are entities, architectures, configurations, and packages. The schema does not represent any data that are internal to the VHDL descriptions, such as components, interface signals, or processes. This kind of information is handled by the VHDL software. This schema granularity is consistent with the black-box tool integration approach, adopted for all design tools (VHDL analyzer and access functions, translators from VHDL to the proof-oriented internal representations, provers).

The schema includes object classes that are specially oriented to the proof process, such as proof-specific translated descriptions, theorems, Boyer-Moore shells, and Boyer-Moore working environments.

All methods offered by PREVAIL-DM are oriented to the application semantics, thus helping to maintain all desired schema-related integrity constraints. Methods help the user to guarantee that all necessary relationships between the objects are created and maintained. Furthermore, methods can only be applied when certain semantic conditions are verified. As an example, the method for removing a VHDL architecture can only be executed if this description has not been proved equivalent against other one.

Library and user management

Design and proof-related objects are stored in two kinds of libraries. System libraries store permanent data, that can be used in several projects. Project libraries contain design data that are particular to given projects. At the beginning of a project, initial data can be retrieved from system libraries. Methods corresponding to design activities, such as creation, compilation, translation to a proof-oriented format, proof, and so on, can be applied only to objects in project libraries. Copy methods allow the transfer of objects between libraries. These methods guarantee the integrity of the data copied into another library, making sure that all necessary relationships to other objects will be also present in the target library.

The environment distinguishes a system administrator from other users. The administrator has full acess to objects and methods. Other users have access to particular project libraries and cannot execute administration methods. User groups can be assigned to different project libraries.

3 CONCEPTUAL SCHEMA

3.1 VHDL source objects

The kernel of the PREVAIL-DM conceptual schema is shown in Figure 1 and considers all VHDL descriptions. A *project library* contains *entities* and *packages*. Each entity may have several architectures. The model allows the designer to designate a single particular architecture as the *specification architecture* for the entity, from which all other *implementation architectures* should be designed. In order to ease the data management, the model restricts a *configuration* to contain component bindings for a single implementation architecture. A specification architecture is assumed to contain no components, because of its behavioral nature. Packages may be used by entities, architectures, and configurations.

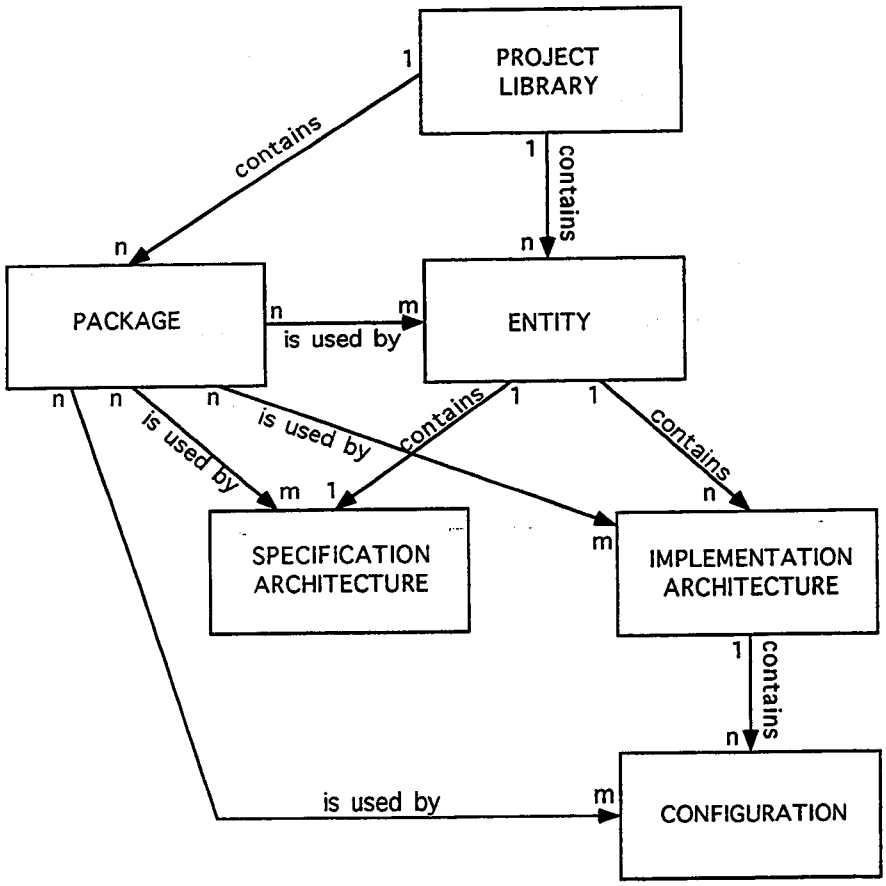

Figure 1: Conceptual schema – VHDL kernel

All objects shown in Figure 1 are source descriptions. For each one of them, there may be an attached *compiled description*, containing only meta-data (user name and compilation date), whereas their "real" compiled contents are stored in the libraries

owned by the VHDL software, that are not integrated into the data base. Each of these VHDL sources may also have an attached *listing* documenting a compilation process.

Table 1 lists the basic methods for handling VHDL descriptions [1]. In this and in all subsequent tables, (u) indicates a method which can be executed by any user, while (a) indicates a method which can be executed only by the system administrator. PREVAIL-DM makes intensive use of the polymorphism feature of the TIDL language. A method with same name may have different implementations for different object classes. Methods *Remove* and *Edit* are typical examples. A package may be removed only if it is not used by any VHDL description, while an implementation architecture may be removed only if it has not been proved equivalent against other descriptions.

for packages, entities, architectures, configurations	Display_Attributes (u) Read (u) Print (u) Copy_to_System_Library (a) Copy_to_Project_Library (u) Create (u) Compile (u) Edit (u) Remove (u)
only for entities	Create_Implementation (u) Import_Implementation (u) Create_Specification (u)
only for implem. architectures	Create_Configuration (u) Import_Configuration (u)
for compiled descriptions	Display_Attributes (u) Remove (u)
for compilation listings	Display_Attributes (u), Remove (u) Read (u), Print (u)

Table 1: Design methods for VHDL descriptions

3.2 Translated objects for the proof process

For supporting the proof process, the schema also includes three object classes corresponding to the internal representations created for the TACHE, LOVERT, and Boyer-Moore provers, as shown in Figure 2. Each instance of these object classes may be attached to a specification architecture, to an implementation architecture, or to a configuration (generally designated as VHDL *bodies*). In the case of TACHE and Boyer-

[1] A complete functional specification of all methods, including the integrity constraints they preserve, can be found in a separate project report [9].

Moore, only a single translated description may exist for each VHDL body. In the case of LOVERT, two translated descriptions may be attached to implementation architectures and configurations, one of them of type "ert" (used as an "implementation" in the proof process) and the other one of type "srt" (used as a "specification").

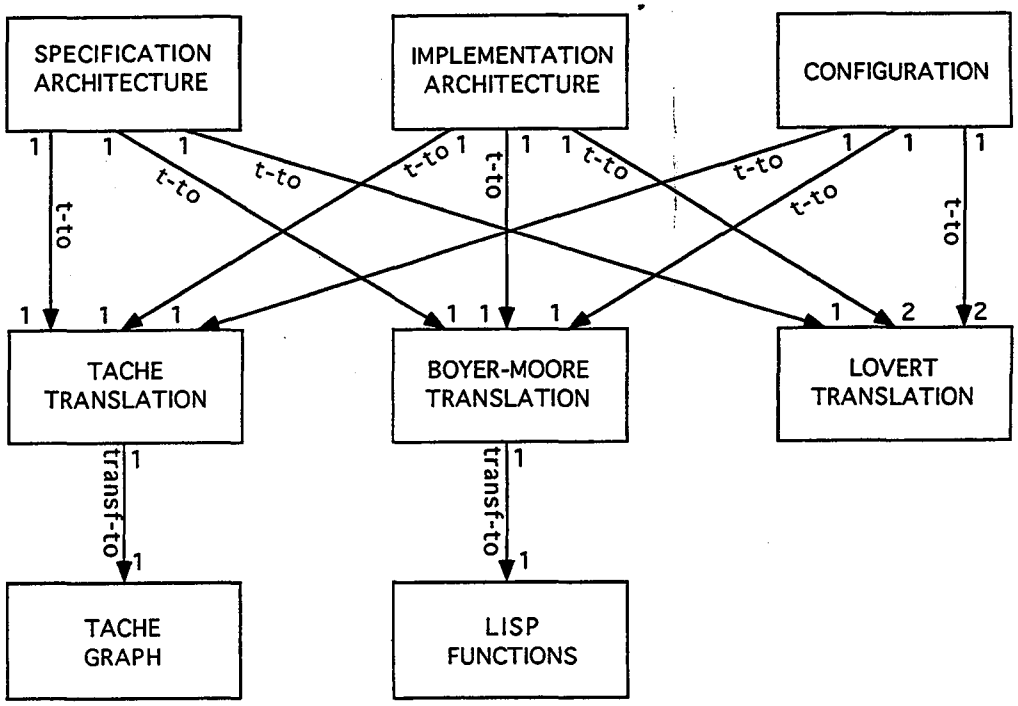

Figure 2: Conceptual schema – translated descriptions

for architectures and configurations	Translate_for_TACHE (u) Translate_for_LOVERT (u) Translate_for_Boyer (u)
for all translations	Remove (u)
for Boyer-Moore translations	Generate_LISP_Functions (u)
for TACHE translations	Generate_TACHE_Graph (u)

Table 2: Methods for the translation process

Table 2 lists additional methods for handling these descriptions. They are created by prover-specific translation methods, that retrieve from the VHDL owned libraries the necessary compiled data corresponding to the VHDL descriptions. In the case of the Boyer-Moore prover, an additional method generates, from the internal representation, LISP functions that are necessary in building the theorem(s) to be proved. In the case

of the TACHE prover, the binary decision diagram generated for a proof can be stored as a separate object and re-used in future proofs. This avoids re-building the diagram, a time-consuming task.

3.3 Equivalence and implication proof

Figure 3 shows extensions to the kernel that are necessary for supporting the equivalence and implication proof. As a result of a successfull proof, *equivalence* relationships may be established between any two implementation descriptions (either implementation architectures without components or configurations of implementation architectures with components). *Implication* relationships may be established from an implementation or a configuration to the specification architecture. In a process of successive refinements, in which each implementation (or configuration) is considered as a kind of specification for a next design step, implication relationships may also directly connect two implementations / configurations. Equivalence and implication relationships are in fact implemented as database objects that are attributed with the name of the proof tool that established the relationship. Listings documenting the proof result may be attached to these *relationship*-objects.

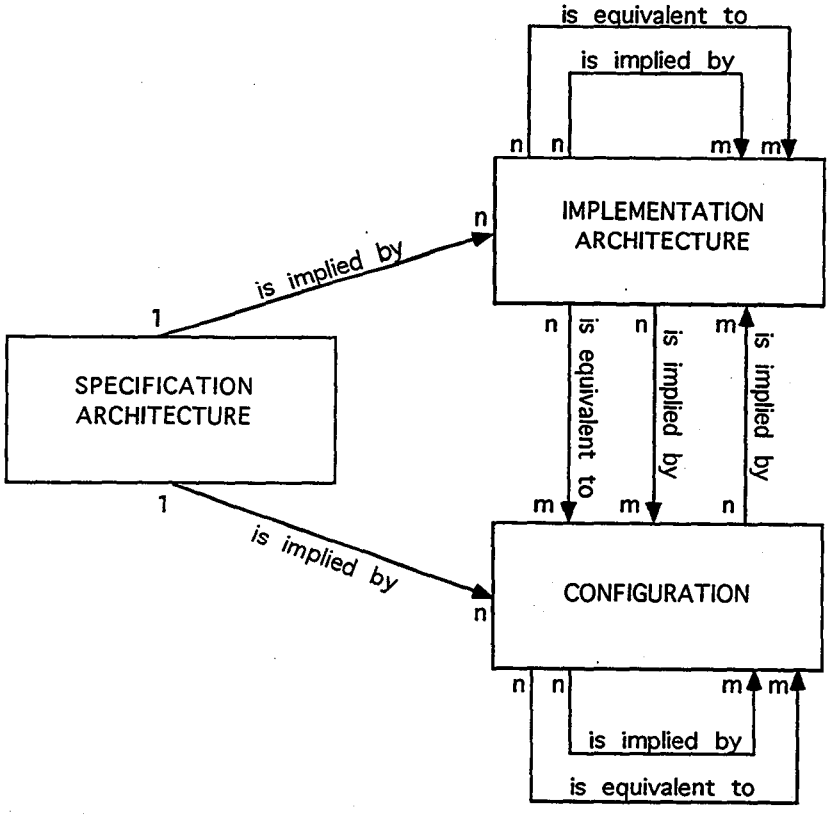

Figure 3: Conceptual schema – equivalence and implication proof

Additional methods, listed in Table 3, include the equivalence and implication proofs by means of each one of the provers, and theorem generation (either stating an equivalence or an implication). Objects of the class *theorem* will be discussed later on. The method *Hierarchical_Proof* (see Section 6.2) builds a new configuration for an implementation architecture X, replacing each implementation architecture I bound to a component of X by the corresponding specification architecture S (but only if it has been proved that S is implied by I).

for architectures and configurations	Prove_Implication_by_TACHE (u) Prove_Implication_by_LOVERT (u) Prove_Implication_by_Boyer (u)
for implementation architectures and configurations	Generate_Theorem_for_Equivalence (u) Generate_Theorem_for_Implication (u) Prove_Equivalence_by_TACHE (u) Prove_Equivalence_by_LOVERT (u) Prove_Equivalence_by_Boyer (u)
for implem.architect.	Hierarchical_Proof (u)
for equivalences and implications	Display_Attributes (u) Remove (u)
for proof listings	Display_Attributes (u), Remove (u) Read (u), Print (u)

Table 3: Methods for the equivalence and implication proof

3.4 Proof of properties

Figure 4 shows an additional extension for supporting the proof of properties. An object *properties* contains one or several assertions on object qualities, described in an appropriate way for the provers. It may be attached to one or several entities, if it contains general properties expected from different design objects. Properties expected from a single design object may be attached to one or several implementation architectures of a same entity. These attachments represent only "expected" properties. An additional relationship is created between an object *properties* and an implementation architecture or configuration for which these properties have been proved. This relationship is implemented as a database object, to which a listing documenting the proof result may be attached. Each object *properties* may have an internal translation for the LOVERT prover, since this is currently the only tool providing capabilities for proving properties.

Additional methods, listed in Table 4, include the creation, edition, attachment and detachment (to / from entities or implementation architectures), translation, and proof of properties. As examples of integrity constraints verified by these methods, an object *properties* cannot be simultaneously attached to entities and architectures and may not be removed nor modified if it has been already proved for a certain description.

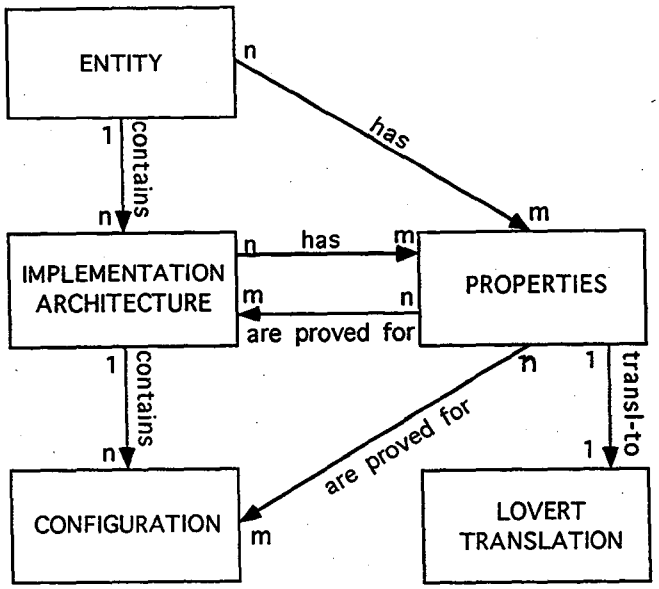

Figure 4: Conceptual schema – proof of properties

for entities and implem.architect.	Create_Properties (u)
for implem.architect. and configurations	Prove_Properties (u)
for properties	Display_Attributes (u) Read (u), Print (u) Edit (u), Remove (u) Translate (u), Proof (u) Attach (u), Detach (u)
for property proof relationships	Display_Attributes (u) Remove (u)

Table 4: Methods for the proof of properties

3.5 Support for the Boyer-Moore prover

Figure 5 shows additional support which is specialized for the Boyer-Moore prover. This tool proves a *theorem* in the context of a given proof environment (a *BM-environment*), which contains *shells* (abstract data types), *functions* (operations on these data types), other theorems, and *axioms* (theorems that are to be considered as already proved in a particular proof context). While proving an equivalence or implication, the user may choose either a user-defined BM-environment or the *bootstrap library*, which contains a pre-defined set of shells, functions, and axioms. As examples of relationships involving these objects, shells are used by functions and theorems, and theorems are attached to two VHDL descriptions (architectures / configurations).

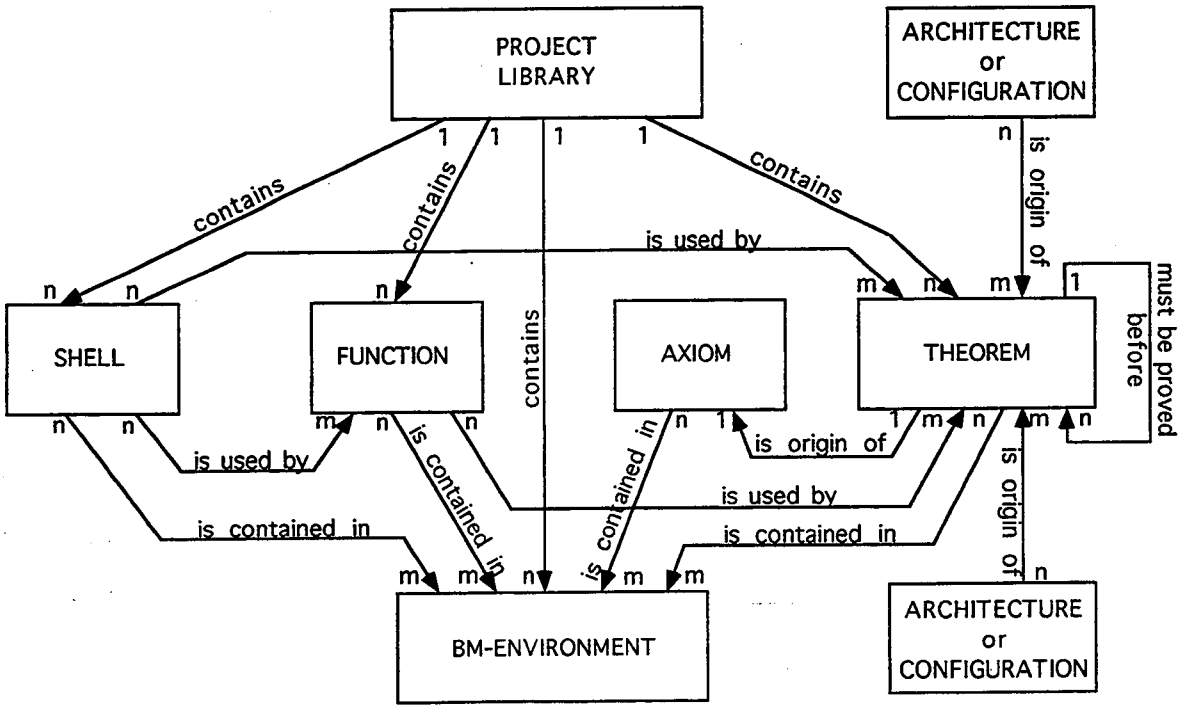

Figure 5: Conceptual schema – support for Boyer-Moore

Theorems may be created in three different situations. Firstly, they are created by the Boyer-Moore pre-processor as a result of a *Prove_Equivalence* or *Prove_Implication* method. Secondly, they may be created, also by the Boyer-Moore pre-processor, by applying a *Generate_Theorem* method, regarding a future equivalence or implication proof. In these two cases, PREVAIL-DM automatically builds relationships between the theorem and the VHDL descriptions originating it. Thirdly, theorems may be interactively created by the user. This provides experienced users with a powerful facility for usage of the Boyer-Moore prover.

PREVAIL-DM offers methods for creating, editing, and removing these objects, as well as including them in / removing them from BM-environments, as listed in Table 5.

It is also possible to store a theorem as an axiom in the context of a particular BM-environment, to restore it back as a theorem, and to prove it (if it has been generated but not proved). Methods guarantee for instance that, if a theorem is included in a BM-environment, the shells and functions it depends on will be also contained in this environment.

for shells, functions, theorems, axioms, BM-environments	Display_Attributes (u) Read (u), Print (u) Remove (u)
for shells, functions, theorems	Create (u) Edit (u)
for theorems	Prove (u) Store_as_Axiom (u)
for axioms	Restore_to_Theorem (u)
for BM-environments	Create (u) Attach_Shell (u), Detach_Shell (u) Attach_Function (u), Detach_Function (u) Attach_Theorem (u), Detach_Theorem (u)

Table 5: Methods for the proof by means of Boyer-Moore

When the user asks for a proof with Boyer-Moore, the pre-processor may generate, depending on the circuit type, not only a single theorem stating the equivalence (or implication) between the output functions of the selected descriptions, but also other auxiliary theorems, that should be proved before the main theorem, as well as a generalized theorem obtained from the main theorem, since the prover is best suited for proving general properties than particular ones. The schema supports the management of the precedence relationships between these theorems.

3.6 Library and user management

Figure 6 shows that design and proof-related objects are stored in two kinds of libraries. *System libraries* store permanent data, that can be used in several projects. *Project libraries* contain design data that are particular to given projects. At the beginning of a project, initial data can be retrieved from system libraries. The *bootstrap library* contains an initial set of shells, functions, theorems, and axioms that may be used with the Boyer-Moore prover. Copy methods allow the transfer of objects between libraries. These methods guarantee the integrity of the data copied into another library, making sure that all necessary relationships to other objects will be also present in the target library.

PREVAIL-DM distinguishes a system administrator from other users. The administrator has full access to objects and methods. Users are attached to particular project

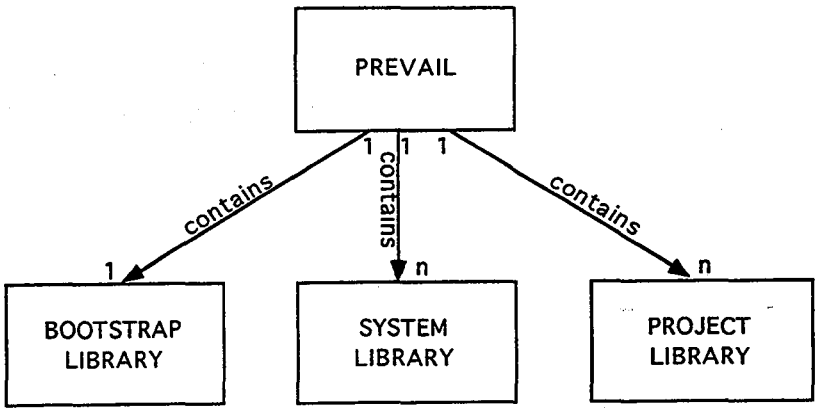

Figure 6: Conceptual schema – libraries

libraries and cannot execute administrative methods. All methods corresponding to design activities, described previously in this section, can be applied by users only to objects in the project libraries the users are attached to. Only four methods may be applied by users to objects in system libraries and in project libraries they are not attached to: *Display_Attributes*, *Read*, *Print*, and *Copy_to_Project_Library*. The administrator may additionally execute methods *Remove* and *Copy_to_System_Library* on objects stored in system libraries, and the method *Copy_to_System_Library* on objects stored in project libraries.

Table 6 shows the administrative methods for library and user management. Users are created outside the scope of particular project libraries. When a project library is created, the administrator may initialize the composition of its user group. The administrator may also attach and detach users to / from project libraries dynamically.

methods applied to the root of PREVAIL-DM	Create_Project_Library (a) Create_System_Library (a) Create_User (a) Remove_User (a)
methods applied to the root of a project library	Attach_User (a) Detach_User (a) Remove_Library (a)

Table 6: Administrative methods

3.7 Examples of methods

Figure 7 gives, as a first example, the complete functional specification of the method *Detach_Shell*, applied only to objects of the class *BM_Environment*. This method does

not encapsulate any existing tool. It is a completely "white-box" programmed method. It can be seen that the method requires user interaction (selection of items from a menu presented by the system according to the context), and that the system verifies if each selected shell may be really detached from the current BM-environment. If positive, the BM-environment is modified and relationships are removed so as to maintain the database consistency.

```
Function:
Detach shells from the current BM-environment

Description:
- user interactively selects shells from a list of shells that
  are attached to the BM-environment
- for each selected shell:
  - remove the relationship from the shell to the BM-environment
  - remove the shell contents from the BM-environment contents
- copy relationships from the BM-environment to other libraries
  are automatically removed

Constraints:
- a selected shell cannot be detached if it is used by functions
  or theorems related to the BM-environment
```

Figure 7: Method Detach_Shell, for BM-environments

Figure 8 gives, as a second example, the complete functional specification of the method *Edit*, when applied to implementation architectures. This method encapsulates a conventional text editor with integrity-preserving pre- and post-activities. It can be seen that, when the user decides to store the new description back in the same object, the system automatically removes some attached objects (such as the compiled description) and copy relationships to objects in other libraries, if they exist, and guides the user to modify existing relationships to other objects, such as packages and properties. When the new description is stored as a new object, the system only guides the user to establish desired relationships to other objects. Different constraints apply to each situation (creation of a new object or modification of an existing one).

As a final example, Figure 9 shows the complete functional specification of the method *Copy_to_Project_Library*, when applied to VHDL sources (entities, architectures, configurations, and packages). This method does not encapsulate any existing tool. In order to preserve the data consistency in the target library, the copy is only possible when certain "father" objects are already present in the target library. Furthermore, certain attached objects and relationships are also copied together.

```
Function:
Edit contents of the current implementation architecture

Description:
- user chooses whether to store the new description in the same
  object or not
- if new description is stored back in the same object:
  - compiled description and listing, translated descriptions,
    LISP-functions, and TACHE-graphs, if existing, are automatically
    removed
  - user may change the relationships from packages to the
    architecture
  - user may change the relationships from the architecture to
    properties
  - objects "property-proven" attached to the architecture (proof
    "relationships" to properties) are automatically removed
  - copy relationships from the architecture to other libraries are
    removed
- if a new architecture is created:
  - user may establish relationships from packages to the architecture
  - user may establish relationships from the new architecture to
    properties already related to other architectures of the same
    entity

Constraints:
- storage of the new description in the same object is possible only
  if:
  - there are no theorems attached to the architecture
  - the architecture has no configurations
  - the architecture has no "equivalence" relationships to other VHDL
    descriptions
  - the architecture is not implied by other VHDL descriptions
- if a new architecture is created:
  - name of new object must be unique among all architectures of the
    entity
- if architecture is stored under the same name:
  - properties cannot be simultaneously attached to entities and
    architectures
```

Figure 8: Method Edit, for implementation architectures

```
Function:
Copy object from the current project library to another project
library

Description:
- the compiled description and listing, if existing, are also
  copied
- if applied to architectures and configurations, the translated
  descriptions, the LISP functions, and the TACHE-graphs, if
  existing, are also copied
- relationships to father objects are also copied (except when
  the father is the project library itself)
- if an "equivalent" description is already in the target library,
  the "equivalence" object (and an eventual attached "proof
  result" listing) is also copied
- if an "implied" description is already in the target library,
  the "implication" object (and an eventual attached "proof
  result" listing) is also copied
- if "properties" already proved for the description is in the
  target library, the "property-proven" object is also copied
- for objects of classes Entity and Package, a son-relationship
  to the target library is established

Constraints:
- user must belong to the user group of the target library, but not
  of the source library
- copy is not possible if object already exists in the target
  library
- name of the copied object must be unique in its context in the
  target library
- if father objects are not yet in the target library, copy is not
  possible; this rule does not apply to "equivalence", "implication",
  "property-proven", and "library" father-objects
```

Figure 9: Method Copy_to_Project_Library, for VHDL sources

4 IMPLEMENTATION ASPECTS

4.1 TIDL realization

TIDL [6, 7] is an object-oriented language offered by the Cadlab framework to define design environments. Besides the schema description (object classes and their attributes, methods applied to the object classes, relationships), TIDL also offers facilities for associating graphical representations to the objects, for defining access management policies, for initializing the database contents, and for "black-box" integrating design tools. The language supports inheritance of methods and attributes between classes and sub-classes, redefinition of methods by sub-classes, and polymorphism. All these features have been used in the implementation of PREVAIL-DM.

There is almost a one-to-one mapping between the PREVAIL-DM object classes presented in the previous section and TIDL classes. Exceptions are due to three modelling problems. Firstly, TIDL-methods are applicable to object classes in a context-independent way. In PREVAIL-DM, however, most design methods may be applied only to the object "copies" that are stored in project libraries. It was therefore necessary to build, for almost all PREVAIL-DM object classes, two particular TIDL-classes, corresponding to the object copies in the project and system libraries, so that most design methods are defined only for the first ones. Secondly, TIDL-relationships may not be attributed. *Equivalence*, *implication*, and *copy* relationships, that must hold attributes, had to be mapped to TIDL-object classes. Thirdly, TIDL does not allow two different relationships between the same objects. This would be needed for relating implementation architectures to properties through an "attachment-relationship" and a "proof-relationship". This latter relationship is thus mapped into an additional TIDL-object class *Properties-Proven.*

Mainly because of the "black-box" tool integration approach and the coarse-grain schema, all classes corresponding to objects that are visible to the user are mapped into database *complex* objects [5]. Most of the object classes are mapped into *unstructured complex objects*, or *uco's*, that have a file as their contents, while some classes are mapped into *equivalence complex objects*, or *eco's*, that have no files as contents. This is the case of libraries, for instance, that serve only to structure the database objects, but are not "objects" in a real sense themselves. Only the object class BM-environment has been mapped into a *structured complex object*, or *sco*, whose contents is a graph of *primitive objects*, or *po's*. These primitive objects, not visible from the user interface, allow the system to find the contents of the shells, functions, axioms, and theorems contained within the BM-environments. As an example, an object of the class *po_shell* instantiates a complex object of the class *shell.*

4.2 Realization of the methods

The Cadlab framework offers a particular black-box tool integration service which is directly available through TIDL constructs. A black-box integrated tool can be associated to a TIDL-method by specifying: database objects to be exported as input files to the

tool; the command for the tool activation, including the exact syntax of optional and mandatory parameters and their data types; and the output files generated by the tool to be imported as database objects. When the corresponding method is selected by the user, the framework automatically calls the tool, handling input and output files as defined in the TIDL script. This service allows a very rapid "encapsulation" of the design tools.

Although its design tools are "black-box" integrated in a general sense (i.e., their input and output files are exported from / imported to the data base, and their codes have not been modified), PREVAIL-DM does not use in fact the specific black-box integration services of the Cadlab framework. This occurs because the environment aims at guaranteeing the schema-related integrity constraints. Therefore, there are always some pre-conditions for executing design tools, such as the existence of given relationships, that must be automatically verified by the methods, as well as integrity-preserving activities that must be automatically executed by the methods after the design tool is executed, such as establishing relationships to other objects.

Therefore, a method does not consist in a single design tool execution. It includes other activities before and after the tool execution. These activities correspond to the most important environment feature, i.e. the automatic verification and maintainance of the integrity constraints. The methods are thus "white-box" integrated, in the Cadlab framework sense [10]. They are C processes that are activated from the framework desktop, receiving as input parameters the identifications of the design object to which the method has been applied and of the current user.

If a method would correspond to a direct execution of a design tool, two alternatives would be possible for the verification of the integrity constraints:

- The user himself/herself would have to manually verify the pre-conditions for the tool execution and establish, after the tool execution, the relationships that preserve the data integrity, through activation of adequate methods. The environment would offer a very weak support for data management, in this case.

- The tools would have to be modified, so that they would contain, in their codes, all necessary verification of pre-conditions and execution of integrity-preserving post-activities. This approach is difficult to implement, because of the need of tool re-writing.

The Cadlab framework also offers an event-handling mechanism which is useful for implementing routines that execute integrity-preserving actions. This mechanism allows the application to define functions that are automatically triggered when certain IDM interface operations are executed or when certain objects or object classes are manipulated (a combination of both kinds of conditions is also possible). Unfortunately, this mechanism cannot be used in the implementation of the integrity-preserving actions of the PREVAIL-DM methods, because the event-functions are triggered on primitive database access operations, not on the activation of TIDL-methods, as would be necessary.

4.3 Access management

The Cadlab framework offers two kinds of services for implementing access management policies. Policies where different users have specific acess rights to different object classes and methods can be implemented by an *Access Manager* [11]. *Client roles*, associated to the cross product *object class x method*, may be defined in the TIDL script of the application. *Clients* have to be dynamically assigned to client roles when new objects are created, in order to gain access to these objects.

As another basic service, the Cadlab framework offers a simple set of *user administration functions* [5] for defining users, handling user membership in user groups, and handling user passwords. These functions have no relation to *clients* and *client roles*. They may help the application to implement the management of users and user groups.

In PREVAIL-DM, to each project library a group of *users* is associated. These users may execute any methods on any objects stored in this library. However, users may execute only "read" methods on objects in system libraries and in project libraries other than theirs. Therefore, the PREVAIL-DM access management policy does not define access rights that depend on the object classes or methods, but on the context where the objects are located.

In the TIDL script of PREVAIL-DM, two roles *Designer* and *Reader* are thus defined. The role *Reader* is associated to "read" methods (*Read*, *Print*, *Copy_to_Project_Library*, and *Display_Attributes*). The role *Designer* is associated to all other methods that can be executed by users on objects stored in project libraries. All users attached by the system administrator to a given project library play the same role *Designer* for all objects within this library. When a user calls one of the methods to which the role *Designer* is associated, the first function executed by the method is to verify if the user really belongs to the user group of the current project library. All users of PREVAIL-DM play the role *Reader* for all objects in all system and project libraries. For the methods to which this role is associated, there is no need of an additional verification of access rights by the methods themselves.

Each time a new object is created within a project library, all users belonging to the user group of the library must be assigned to the role *Designer* with respect to this object, in order to gain access to the methods defined for the corresponding object class. Furthermore, all users of PREVAIL-DM must be assigned to the role *Reader* for this object. This dynamic role assignment is implemented by one of the functions offered by the Access Manager.

As another consequence of the framework features, each time a new user is attached to a project library, he/she must be assigned to the role *Designer* for each of the objects already existing in the library. Inversely, when a user is detached from a project library, the user must loose the role *Designer* for each of the objects already existing in the library.

As already stated in Section 4.1, in the Cadlab framework the executability of methods defined for a certain object class does not depend on the context where objects of this class are situated. This resulted in the definition of special classes for objects located in the system libraries, so that for these classes only "read" methods are available for the

users.

Methods that can be executed only by the PREVAIL-DM *administrator* have no client role associated in the TIDL script. As a consequence, only the framework administrator (pre-defined role *dba*) can execute them. The *dba* thus directly implements the PREVAIL-DM *administrator*.

4.4 User interaction

The PREVAIL-DM user interface is implemented by two different mechanisms. The *main* user interface of the application is automatically available through the framework *desktop* [8], which displays the database contents and allows the user to browse through the objects and relationships. For each object there is pop-up menu containing all methods available for the object class. This desktop is based on Motif and offers many other control facilities, described in [8]. Access to objects and methods is possible according to the client roles the current user plays. As a result, the PREVAIL-DM users can navigate through all libraries and select any "read" methods. The execution of "write" methods on objects located on libraries to which the user has no "write" access will be refused by the method execution itself, and not by the framework mechanisms.

Furthermore, each design method builds its own user interface. Many methods have to interact with the user, for displaying menus with lists of objects, receiving menu selections, displaying messages, receiving textual input, and so on. These method interfaces are also based on Motif, so that the PREVAIL-DM user has a comfortable and homogeneous interaction with the whole environment.

5 ANALYSIS OF THE ENVIRONMENT

5.1 Support for VHDL and schema granularity

PREVAIL-DM implements a coarse-grain VHDL-based schema, where the main objects are entities, architectures, configurations, and packages. As a restriction, the schema considers that a configuration is related to a single architecture, and not to an entity. This restriction eases the data management.

All design tools are black-box integrated. This integration approach considerably eases the development of the environment prototype, since the code of tools is not touched.

The schema does not represent any design data that are internal to VHDL objects, such as interface signals, attributes, processes, functions, data types, etc. In particular, components within architectures are not modelled. Therefore, the environment does not have knowledge about the structural decomposition of the architectures and the binding of these components to other entities / architectures. All internal data of VHDL objects, including the information about this structural decomposition, is handled by the VHDL software and by the design tools (analyzer, translators, provers).

Because of the schema granularity and tool integration approach, the environment is not responsible for verifying any integrity constraints related to design data that are internal to the VHDL objects. These constraints are verified by the design tools.

The compiled representations created by the VHDL analyzer are not stored as database objects. The environment only maintains a record of which VHDL sources have been successfully compiled, by attaching to them objects of the class *Compiled_Description*, that contain only meta-data for management purposes.

5.2 Support for equivalence, implication, and properties proof

The schema includes all auxiliary objects that are needed in the proof process, when using the TACHE, LOVERT, and Boyer-Moore provers: translated descriptions obtained from the VHDL sources and used as input for the provers; listings documenting the proof results; theorems, shells, functions, and axioms; and working environments for the Boyer-Moore prover.

The coarse-grain approach is used also for these objects. Their internal data are handled only by the design tools. The environment is responsible for maintaining the relationships between these objects and from them to the VHDL objects.

The model includes objects of a class Properties, for storing properties of entities and architectures that are to be proved.

The model distinguishes a particular architecture as the entity specification. Only one such architecture may exist for each entity. For this architecture there are no configurations, since it is supposed that it contains no components. Other architectures are considered as implementation architectures.

The model includes relationships specially suited for supporting the equivalence, implication, and properties proof:

- implication from an implementation / configuration to the specification;
- implication from an implementation / configuration to other implementation / configuration;
- equivalence between two implementations / configurations;
- properties proved for an implementation / architecture.

5.3 Design data integrity

Two main requirements in the PREVAIL-DM development were the "black-box tool integration approach", adopted because changing the tool codes would be a too great effort, and the automatic verification of integrity constraints, so that the environment really offers more than the already available PREVAIL system.

In a design environment, we can identify three levels of integrity constraints that must be verified during a design process:

- low-level, fine-grain *data constraints*, such as the compatibility of signal data types;
- high-level, coarse-grain *data constraints*, such as equivalence relationships between VHDL architectures;
- high-level *methodology constraints*, such as the achievement of design qualities like cell area or maximum delay.

Design constraints at the first level are normally verified by using tools like HDL compilers, simulators, and verifiers. In the PREVAIL context, the VHDL analyzer verifies syntactical conditions, the translators verify if the proper VHDL subset is being used, and the provers verify if the design data that are "internal" to the VHDL descriptions have the desired qualities.

It would be meaningless to build a framework-based environment that verify these constraints at the database level (either by the schema or by the methods), since they can be much more efficiently verified by the already existing tools. This is in fact the main reason for adopting a coarse-grain conceptual schema.

The design data integrity constraints that are automatically verified and maintained by PREVAIL-DM are at the second level and regard the relationships between the coarse-grained objects of the schema. As examples, it can be remembered that:

- methods for creating objects enforce or guide the user to establish the correct relationships to other objects;
- objects can be removed or modified only if they do not participate in relationships that must be maintained, such as equivalences;
- copy methods guarantee that an object is copied into another library in such a way that data integrity is maintained (for instance, other objects and relationships are copied together).

Design constraints at the third level are methodology-specific and their verification should not be integrated nor into the design tools neither into the conceptual schema. Their formalization is tipically used for controlling the design flow. This is the objective of a further development in the PREVAIL context, discussed in Section 6.1.

5.4 Environment extensibility

PREVAIL-DM offers the extensibility capabilities that are intrinsic of the Cadlab framework.

The schema is defined by means of a TIDL script, which specifies object classes, relationships, and methods. This script may be extended by new definitions of classes, relationsips, and methods. The TIDL compiler automatically verifies the consistency of the extensions with regard to the already implemented script. The already existing database contents is preserved.

It is not possible however to remove definitions of existing classes, relationships, and methods.

The TIDL script in fact does not contain the methods themselves, only references to functions that implements them. As already explained in Section 4.2, the Cadlab black-box integration services cannot be used for integrating new tools into the environment. A "white-box" method that encapsulates the new design tool must be written, if we want to preserve the integrity constraints.

5.5 Support for experienced users

PREVAIL-DM supports the use of formal provers by experienced designers. In the case of Boyer-Moore, for instance, the systems supports the definition of shells, functions, and theorems and their integration into a working environment for the prover, as well as the proof of isolated theorems.

These features would be of no use for designers with little experience on formal proof. These designers need in fact a proof process as automated as possible, following a system-guided interactive dialogue for giving the information which is needed for the proof, as in the currently available PREVAIL environment.

Such a kind of process automation can be also obtained in this environment, when design methodology management is added. In this case, task sequences that correspond to sequences of the available methods can be defined and automatically executed.

5.6 Features not found in the environment

Other useful features expected from integrated environments and / or supported by design frameworks are are not implemented in PREVAIL-DM:

- version management;
- configuration management;

- design methodology management;
- common internal formats;
- intertool communication;
- uniform user interface for all design tools.

The last two features from the list above have not been implemented because of the black-box tool integration approach. Configuration management would be only useful if a fine-grain schema had been implemented, so that the environment would be responsible for handling components inside structural descriptions and their bindings to other entities / architectures. In PREVAIL-DM, configuration management is realized by the VHDL software and is not integrated into the environment.

Version and design methodology management are goals to be pursued in next environment releases, as discussed in the Section 6. A common internal format for all design tools is still a research theme in the field of formal hardware verification.

5.7 Features of the Cadlab framework not used in the environment

The following features are supported by the Cadlab framework but have not been used in the implementation of PREVAIL-DM:

- version management;
- black-box tool integration mechanisms;
- event handling;
- support for modelling complex structured objects;
- intertool communication services.

As already stated, version management is a goal of a future environment release. Although design tools integrated in PREVAIL-DM are "black-box" integrated in a general sense, the reasons for not using the specific Cadlab black-box tool integration mechanisms have been discussed in Section 4.2. The same section explain why the event handling mechanisms have not been used. Intertool communication services have not been used because of this black-box tool integration approach adopted in the environment. Complex structured objects have been used in a very limited extent for implementing BM-environments. This feature would be used more extensively if a fine-grain schema had been adopted.

5.8 Environment features that cannot be used by the current PREVAIL tools

Some of the features implemented in PREVAIL-DM cannot be used by the proof tools currently available in the PREVAIL environment. Nevertheless, these features have been implemented in the conceptual schema because an evolution of the proof capabilities in their direction is expected or desired in the near future. There are two basic reasons that explain why some features cannot be used.

Firstly, the proof tools, in fact, handle currently only very simple VHDL descriptions. The provers only accept fully configured architectures (architectures already containing the binding from their components to other entities / architectures), so that all proof relationships envolving VHDL configurations cannot be used. Furthermore, the tools search all necessary VHDL data types and functions in a single package, so that the definition of multiple packages and their attachment to VHDL descriptions will not be used for the moment.

Secondly, in the case of the implication proof there is currently no experience or established proof capabilities that correspond to this environment feature.

6 FUTURE WORK

6.1 Design methodology management

Design methodology management [1] is an additional service supported by design frameworks. It has multiple possible goals:

- automatically execute user-defined task sequences, eventually specified in a hierarchical way;
- associate multiple tools to a task and choose (either automatically or guided by the user) the most promising one in a given situation;
- verify if tasks have achieved desired goals (for instance object qualities), and if not
  - display possible alternatives to the user, or
  - automatically execute another task sequence alternative;
- evaluate design alternatives, either using some evaluation tools or executing the alternatives and comparing the results;
- in conjunction with data management capabilities, automatically guarantee design data consistency as the tasks are executed;
- store task sequences defined on-the-fly and allow their re-execution, eventually modifying tool or object selections.

When performing a complete proof process in the PREVAIL environment, some task sequences may be easily identified. As an example, for an equivalence proof using Boyer-Moore the following sequence is necessary:

1. select an entity and verify if it has been successfully compiled; if not, compile it;
2. select two architectures / configurations and verify if they have been successfully compiled; if not, compile them;
3. identify the circuit types of the chosen descriptions, in order to choose the most appropriate prover;
4. translate the descriptions into the input format for the chosen prover – a preprocessor generates the theorem to be proven;
5. relationships between the theorem and the two descriptions must be established;
6. if the proof succeeds, the theorem is marked as proved, and an equivalence relationship between the descriptions is established.

Since it is expected that this sequence will be repeated very often, the application to its execution of some of the above features listed for a design methodology management mechanism would be valuable.

As already mentioned in Section 5.5, these features are very useful for non-experienced users that follow methodologies defined by a "methodology manager". They are however also useful for experienced users, that may define their own methodologies and try many alternative proof strategies.

It is also clear that the execution of a task sequence must conform to a data schema, and specially to data integrity constraints, as implemented for instance in the PREVAIL-DM environment.

Methodology management adds another level of integrity control to a design environment, which is related to the task sequence integrity, as presented in Section 5.3.

A prototype of a design methodology manager, also based on the Cadlab framework, is now under design. It will implement some of the features listed above. In particular, it will bring together design methodology management mechanisms specified for the STAR framework [12], under development at the University of Rio Grande do Sul, at Porto Alegre, Brazil, with the data management capabilities of PREVAIL-DM.

6.2 White-box tool integration

In a general sense, white-box tool integration means that the tool code "knows" the data base schema and directly manipulates data base objects and relationships.

This tool integration approach has two consequences. Firstly, as already mentioned in Section 4.2, the tool can automatically verify and preserve the design data integrity, without the need of writing' complex methods that encapsulate the tool. Secondly, this approach is also consistent with a fine-grain schema, which models design data that are internal to VHDL and other proof-related objects.

If a tool already exists, which makes such fine-grain integrity verification (such the VHDL analyzer, which verifies the integrity between entities, architectures, configurations, and packages), the automation of this verification in the data management system does not add too much value to the environment, making the schema unnecessarily more complex.

There are cases, however, when only a white-box integrated tool, with knowledge of the schema, can add a valuable feature to the environment. This is the case of the method *Hierarchical_Proof* proposed in PREVAIL-DM for VHDL architectures. This method verifies, for each component C of an architecture A, if the architecture X (of another entity E) bound to C (either by means of a configuration declaration inside A or of an external configuration body attached to A) has been already proved against the specification architecture S of E. If X has been proved against S, then the method replaces X by S in a new configuration it creates for A. This strategy replaces an eventually very deep hierarchy inside A by a configuration which is only one-level deep and which binds a behavioral specification to each component of A. This replacement may ease the proof of another object containing an instance of A.

This method accesses internal design data information of VHDL architectures and

configurations as well as schema information about implication proofs already performed. It remains as a further goal of PREVAIL-DM the development of such specialized "white-box" integrated design methods.

6.3 Version management

PREVAIL-DM offers the version management facilities that are intrinsic of the VHDL language. Each *entity* may have several *architectures*. Architectures of a same entity may correspond to different design *alternatives*, to representations (*views*) at different design abstraction levels, and to successive refinements (*revisions*) of a given alternative at a given abstraction level. There is no means of distinguishing among these three situations in VHDL, as opposed to more powerful version management models [12, 13].

Although the support for the identification of design *alternatives* and *views* is out of the scope of PREVAIL-DM because of VHDL, it is possible to implement a *revision* control for each architecture, and even for each configuration. This would have an immediate benefit in avoiding an explosion in the number of architectures for each entity, since in the current environment each design change must be stored in a different architecture.

PREVAIL-DM has no intrinsic versioning capabilities for other object classes, like packages, theorems, shells, and so on. Revision control may be also implemented for these classes, having a similar impact on the explosion of design objects.

The Cadlab framework already supports a versioning mechanism for database objects [5]. This support can be used to build a revision control for all PREVAIL-DM object classes. This control impacts the design methods in different ways.

Always when a method fetches a design object, an object version must be selected. It is possible to implement a default mechanism, so that the most recent version is always selected, except when the user explicitly wants to select an old version.

Always when a method creates an object, it must be decided whether this object corresponds to a new version of an already existing object, to the replacement of the most recent version of the object (if the version model allows this possibility), or to the first version of a new object. There are two possible solutions for implementing this choice:

- new methods are introduced for creating versions for already existing objects – in this case, the default mechanism can correspond to the creation of a new version, and the user must explicitly indicate when the new representation should replace the most recent version;

- in the already available methods, if the user indicates the name of an already existing object, a new version is created for it (or the most recent version is replaced); if the user indicates a new name, a new object is created.

These considerations does not apply to certain objects that are not created under explicit user control, but as an "indirect" consequence of the method execution, such as compiled descriptions and listings, "equivalence" and "implication" objects, and son on. In these cases, the method must be implemented with an already "embedded" solution.

In the case of a compilation listing, for instance, where only an object of the class may be attached to a VHDL description, the creation of a new object is not possible.

It must not be overseen that the version control has a great impact also on the relationships between objects. If two objects have many versions each, a relationship between them must be established between selected versions. This feature is also supported by the Cadlab framework. Integrity constraints must be therefore changed according to this fact. In the above mentioned situation, for instance, only one object of the class *Compiled_List* may be attached *to each version* of a VHDL description.

From the above discussion, it is clear that the implementation of even a very simple revision control mechanism represents a big re-design and re-programming effort.

References

[1] D.S. Harrison, A. Richard Newton, R.L. Spickelmier, and T.J. Barnes. Electronic CAD frameworks. *Proceedings of the IEEE*, February 1990.

[2] D. Borrione, L. Pierre, and A. Salem. Formal verification of VHDL descriptions in the prevail environment. *IEEE Design & Test of Computers*, June 1992.

[3] IEEE, New York. *IEEE Standard VHDL Language Reference Manual*, 1988.

[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] CADLAB. CADLAB object management system release 2.3: IDM – user's guide. Technical report, CADLAB, Paderborn, 1991.

[6] K. Groening et al. From tool encapsulation to tool integration. In F.J. Rammig and R. Waxman, editors, *2nd IFIP International Workshop on Electronic Design Automation Frameworks*. North-Holland, 1991.

[7] CADLAB. CADLAB tool integration description language release 3.0: TIDL user's guide. Technical report, CADLAB, Paderborn, 1990.

[8] CADLAB. CADLAB desktop release 1.0: User's guide. Technical report, CADLAB, Paderborn, 1990.

[9] F.R. Wagner. PREVAIL-DM: Functional specification of the methods. Technical report, IMAG / ARTEMIS, Grenoble, 1992. (Internal project report, unpublished).

[10] CADLAB. CADLAB tool integration description language release 3.2.6: TIDL user's guide. Technical report, CADLAB, Paderborn, 1990. (Preliminary).

[11] CADLAB. CADLAB access manager release 1.0: AM – user's guide. Technical report, CADLAB, Paderborn, 1990.

[12] F.R. Wagner, L.G. Golendziner, J. Lacombe, and A.H. Viegas de Lima. Design version management in the STAR framework. In *3rd International Workshop on Electronic Design Automation Frameworks*, Bad Lippspringe, Germany, 1992. IFIP.

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