UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Semantic extraction for the automatic verification of VHDL descriptions.

Tiziana Margaria

Fakultät für Mathematik und Informatik
Universität Passau
Innstr. 33
D-94032 Passau (Germany)
tel: +49 851 509.716
fax: +49 851 509.346
tiziana@fmi.uni-passau.de

Maurizio Griva, Riccardo Tesio

Dip. di Automatica e Informatica
Politecnico di Torino
10129 Torino
Italy

MIP-9411
October 1994

Abstract

We present V2O, an automatic semantic extraction tool for a subset of VHDL, which takes care of the axiomatization of pairs of VHDL descriptions whose behavioural equivalence is to be checked within the OTTER based verification system. The implementation is based on enriched axiomatization criteria in terms of gate level and RT level specific primitives.

Contents

1 Introduction

A completely integrated and structured verification methodology for real life designs at the Register Transfer level requires circuit descriptions that combine well-defined semantics with the full power of a RT level Hardware Description Language. VHDL is a standard [IEEE88] language which is frequently used in design environments, mostly in connection with commercial simulators. However, the interest in automating proofs of correctness of designs described in this language is increasing significantly.

In this paper, we present V2O, an automatic semantic extraction tool for a subset of VHDL, which takes care of the axiomatization of pairs of VHDL descriptions whose behavioural equivalence is to be checked within the OTTER based verification system. The design and realization of the tool has profited of the experience gained in the axiomatization of a smaller HDL, ART∗ [PrGa85]. ART∗ is a non procedural Register Transfer level language with well defined semantics, that allows behavioural, functional, structural, and multi-level descriptions, for both asynchronous and synchronous hierarchical design styles. Its axiomatization criteria in terms of gate level and RT level specific primitives have been presented in [Marg92], and they have been enriched and tailored for the VHDL language.

Related Work

Among the successful efforts in this direction, the PREVAIL [BoPS91] approach is closest to ours. First, syntactic subsets of the language appropriate to describe zero-delay combinational circuits, generic repetitive circuits and categories of synchronous sequential circuits have been identified. Descriptions are then automatically translated in the input format of either a tautology checker, or the Boyer-Moore induction-based theorem prover, or LOVERT [BEFP89], a specialized tool for RT level verification. Different types of circuits requiring different verification strategies are verified with different tools, with little overlapping of the competences.

On the contrary, we started from the capabilities of the OTTER general purpose theorem for First-Order Logic with equality, which offers a rich set of verification strategies. This led to HDL independent axiomatizations in First-Order Logic of the semantics of descriptions up to an extended RT level. We are now able to accept descriptions in ART∗, and in a subset of VHDL which is very close to the subset accepted by PREVAIL. Different types of circuits just require a different formalization of the goal for the OTTER prover, while the selection of the proof strategy is done by linking sets of commands in the header of the OTTER file. Automatization criteria for both tasks have been found, but not yet implemented in the prototype.

The paper is organized as follows: Section 2 presents the homogeneous semantic framework for the axiomatization in First-Order Logic of descriptions in an HDL, Section 3 illustrates the application domain, i.e., the VHDL description language and the OTTER general-purpose theorem prover for First-Order Logic with equality, Section 4 ilustrates the automatic semantic extraction phase as it is performes by means of the V2O translator, and Section 5 presents an examples and our results on several benchmark circuits.

2 The *homogeneous* semantic framework

Our notion of correctness of a circuit's implementation with respect to a specification is based on the following notion of behavioural equivalence:

> *Two descriptions D_1 and D_2, with boolean primary inputs $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$, respectively, are called* behaviourally equivalent, *$D_1 \approx^B D_2$, iff, from the respective reset state, they produce the same output for each vector of input values v, i.e. whenever the input variables are initially assigned equally.*

Thus our specification-implementation relation is defined by:

$$Spec(C) \vdash Impl(C) \text{ iff } D_{Spec(C)} \approx^B D_{Impl(C)}.$$

The verification method we propose is based upon the idea that the behavioural equivalence between two descriptions can be characterized as the consistency of a specific derived theory of First Order Logic. This theory is given by the combination of the axiomatizations for the two descriptions, together with the *thesis*, i.e. a formalization of the claim that both descriptions are behaviourally equivalent. The method is summarized in Figure 1.

Altogether, the verification in our framework proceeds in three steps:

- axiomatizing the specification and the implementation descriptions in First-Order Logic
- enriching the axiomatizations of the two description by adding equations that formalize input/output equivalence, and
- checking the consistency of the enriched set of axioms.

Once the semantic extraction criteria have been individuated, the first two steps are simple mappings from the HDL descriptions into the corresponding logic formulæ, and are achieved by translation. The third step can then be carried out entirely automatically by means of the OTTER general-purpose First-Order Logic theorem prover [McCu90].

The semantics of hardware descriptions is given in terms of First-Order theories, i.e. as axiomatizations in First-Order Logic with equality.[1] These axiomatizations are composed of an *equational* characterization of all the HDL language constructs that express the semantics of hardware specific function and predicate symbols, and some additional equations characterizing the specific properties of the description under consideration.

[1] A similar approach can be found in [Evek85]

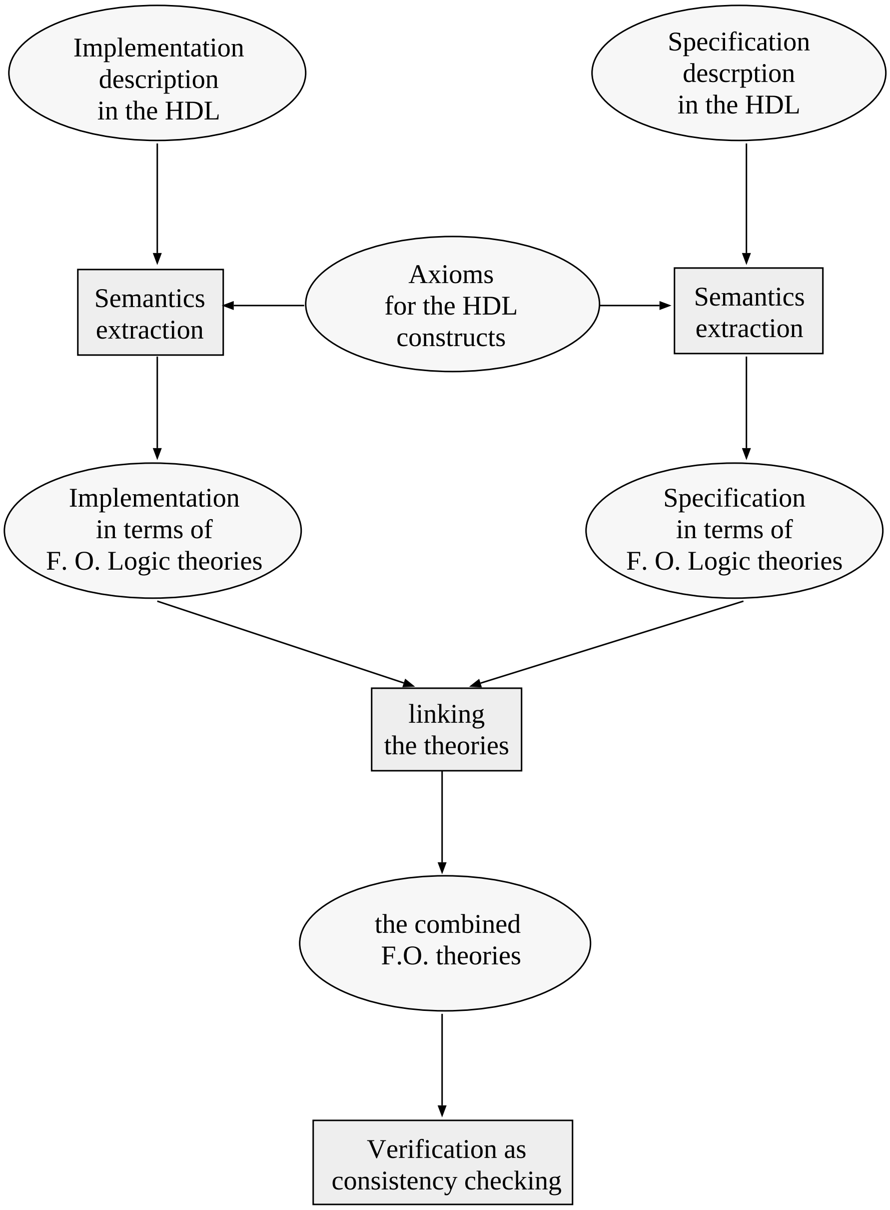

Figure 1: The verification process

Axiomatization of an HDL The axiomatization of an HDL consists of an axiomatization of pure *First-Order Logic*, together with three sets of equations characterizing the constructs of the *gate level*, *RT level* and the HDL under consideration ([Marg91]):

1. First-Order Logic with equality: its axiomatization is built in the OTTER theorem prover,

2. gate level: we define the gate-level semantics on the basis of a *boolean algebra*, a *time model*, and a *substitution mechanism*. By substitution of equals for equals

in the circuit equations, combinational subnetworks are transformed into boolean expressions. The access to past values of signals enables us to model sequential circuits (for asynchronous HDLs it just denotes a before/after relation),

3. RT-Level: we build the RT-level semantics as an extension of the gate-level one, to yield a fully compositional semantics encompassing both levels. *List selector* and *constructor* operators are the sole additional primitives needed to deal with bit_vectors and matrices of booleans, that represent registers, memories, PLA's. *Conditional* primitives to express control statements are defined starting from gate level control primitives,

4. HDL-specific theory: we associate a First-Order Logic formula to each language primitive, to express its intended semantics in terms of the previously defined semantic kernel.[2]

The four portions of the axiomatization for an HDL specify the meaning of disjoint sets of constructs. The consistency of the whole axiomatization follows then from their individual consistency. In this highly modular approach, the consistency of the logic, gate-level and RT-level axiomatizations can be verified once and forever. Afterwards, the consistency of the axiomatization of a particular HDL is simply a consequence of the consistency of the HDL-specific axiomatization.

We already followed this idea and used the OTTER theorem prover to show that the axiomatization of ART∗ given in [Marg91] is consistent. To achieve now a consistent axiomatization for a new HDL, like VHDL, only the consistency of the VHDL-specific axioms of the semantics has to be shown.

Axiomatization of a Description The axiomatization of a description requires to enrich the axiomatization of the HDL by equations that completely formalize the *functionality* of that description. To this aim, to each statement is associated a suitable nested composition of the corresponding semantic primitives. The resulting equation set, which consists of boolean operators, list operators, accessors to past values and nested if constructs, is still easily readable.

This axiomatization can be derived straightforwardly by translation of the description into the logic.

Formalization of the Thesis In order to establish the link between a pair of theories it suffices to constrain them to share the same input variables and to relate the initial states of the machines. The requirement that the desired equivalence relation must then hold for all input configurations (handled symbolically), constitutes the goal. Subsequently, the proof of functional equivalence is achieved via consistency checking on the resulting global theory [MaSt93, MaSt93b].

[2]In practice, a set of simplification rules and basic equivalences or useful reorderings are individuated, to become candidate *theorems* in the semantic theory. These theorems can automatically be verified by means of OTTER.

Once the logical theories are expressed in terms of the syntax of the OTTER prover, the proof, which only involves the semantic level, can be done entirely automatically (see Section 3.2).

In thie next sections, we consider the case of synchronous sequential circuits, whose specifications and implementations are described in the VHDL language and whose verification is carried out within the OTTER-based verification system.

3 Application

3.1 VHDL as source language

As VHDL has a rich set of operators, exceeding the ones already defined for a standard RT level HDLs, the axiomatization has required first to introduce the needed new fundamental semantic primitives, and then to implement the mapping criteria from VHDL descriptions into the set of First-Order Logic equations for OTTER

Axiomatization of a subset of VHDL A nucleus of additional non strictly RT level primitives defines natural numbers in terms of Peano's axioms and an arithmetic on natural numbers based on addition, subtraction and multiplication. The extension to integers enables us to automatically deal with VHDL specifications at the integer level. Conversion functions transform natural numbers into bitvectors, integers into bitvectors in two's complement notation, and viceversa. The lack of a canonical form to represent integer expressions prevents us from handling operations on integers always entirely at the integer level. Then, a conversion to bitvector format, in either two's complement or module and sign representation, enables us to perform the desired comparisons and operation at this lower level.

While VHDL models bitvectors essentially as an array, i.e., as a data structure in which each element can be directly accessed in a single step, OTTER models bitvectors as lists, for which only the head is directly accessible. For access efficiency reasons, bitvectors are then stored as lists with Least Significant Bit left, since operations usually chain from LSB to MSB.

VHDL types not available in ART$*$, like enumeration types and records, are naturally modelled as lists of objects. Objects can be simple constants in an enumerative type, or other lists in complex records. Access to single record fields or the definition of subtypes are done via the list selector and constructor RT level functions.

Axiomatization of VHDL descriptions The axiomatization of the descriptions can be achieved in a procedural way, by means of automatic semantic extraction with the V2O tool, We do not explicitely distinguish between combinational and sequential descriptions: th eevaluation function of all signals and variables takes time and default values into consideration.

For a successful verification, VHDL descriptions are required to satisfy the following conditions:

- *combinational* portions of a circuit are handled within a zero-delay model, so that the VHDL timing constructs (after, wait, ...) are ignored. No restriction is posed on the presence of combinational loops in the descriptions: they are in fact detected during the verification, at the OTTER level.

- *sequential* circuits do not admit resolved signals. We accept only simple processes, with sensitivity list, and without cycles. Variables can be assigned only once within a process, and if the process has memory the user must ensure that the process is activated at most once for each stimulus. One possible simple criterium, although more restrictive than actually required, is the following: a process with memory is accepted only if it has only one activation signal, this signal is a port of the entity and this port is linked to a test signal.

For each signal or variable written by the process, the translation into logic equations uses the following information: denoting with AS the set of signals in the sensitivity list, and with RS the set of read signals, and with RV the set of read variables,

- $RAS = RS \cap AS$ is the set of read signals also in the sensitivity list,

- $RNS = RS \setminus AS$ is the set of read signals not in the sensitivity list,

- RBV is the set of read variables which are assigned a value em before the current instruction

- $RSV = RV \setminus RBV$ is the set of read variables which are assigned in the current instruction or later.

Denoting with *v(id, t)* the evaluation function that returns the value taken at time t by signal or variable *id*, for each *signal* written by the process we require that either all the signals inAS are directly connected to test signals, and all the signals in RNS are driven by other processes, or the user must ensure that the signals of the two sets vary as a consequence of disjoint stimuli. Then we generate in the most general case the following two equations:

$$
\begin{aligned}
v(S,i) = if(event(AS,i) \\
&\quad then\ \ f_S(v(RAS_S,i), v(RNS_S,i-1), v(RBV_S,i), v(RSV_S,i-1)) \\
&\quad else\ \ v(S,i-1)
\end{aligned}
$$

$$v(S,0) = value.$$

For each *variable with memory* the equations have the form

$$
\begin{aligned}
v(V,i) = if(&event(AS,i)\\
&then\ \ f_V(v(RAS_V,i), v(RNS_V,i-1), v(RBV_V,i), v(RSV_V,i-1))\\
&else\ \ v(V,i-1)
\end{aligned}
$$

$$v(V,0) = value.$$

In the case of combinational signals and variables, since $RS = RAS$ and $RV = RBV$, the above equations take the simplified form

$$v(S,i) = f_S(v(RS_S,i),\ v(RV_S,i))$$

$$v(V,i) = f_S(v(RS_V,i),\ v(RV_V,i)).$$

Single clock assumption The model can be slightly simplified for FSMs, where a single clock signal often activates all registers and flip-flops. All sequential devices are therefore driven by the same clock edge: a D type flip-flop would be modelled as

$$
\begin{aligned}
v(Q,i) = if(&event(Clk,i)\\
&then\ \ if(v(Clk,i) = 1)\ then\ v(D,i-1)\ \ else\ \ v(Q,i-1)\\
&else\ \ v(Q,i-1)
\end{aligned}
$$

If the stimulus for the assignment of the input values is taken to coincide with the passive clock edge, we have that $v(Clk,i) = \neg v(Clk,i-1)$, $V(Clk,0) = 1$, and $\forall i,\ event(Clk,i) = true$, so that we write

$$v(Q,i) = if(v(Clk,i) = 1)\ \ then\ \ d(D,i-1)\ \ else\ \ v(Q,i-1).$$

If inputs assume exactly on the active clock edge the new stable value determined at the preceding passive clock edge, passive edges need not be modelled anymore, and we have the classical model of the flip-flop:

$$v(Q,i) = v(D,i-1).$$

A similar model can be derived for the other types of flip-flops and for registers. Being mostly interested to the verification of several classes of Finite State Machines, the generated equations are in the latter form.

3.2 The OTTER-based verification system

OTTER is a resolution-based theorem prover for First-Order Logic with equality developed at the Argonne National Laboratories [McCu90]. It is compactly coded in C (the executable file occupies only 120 Kbytes), and it is portable to a large variety of computers. We have it currently running on SUN *SparcStations II* under UNIX and on PC under MS-DOS.

After accepting either First-Order Logic formulæ or clauses as input, OTTER proves its goal by *refutation*. The supported inference rules include *binary resolution*, *hyperresolution*, *UR-resolution*, and *binary paramodulation*.

In our system, proofs proceed in three steps: first, the semantic expressions for the specification and the implementation are rewritten to yield functions of primary inputs only. Second, the resulting expressions are *simplified* and *canonicalized* to a nested if form Third, the proof by refutation is executed according to the enforced proof strategy.

The choice of the proof strategy is left to the user. A wide selection of options concerns the basic inference rules already mentioned, not only with respect to their enforcement, but also to control their application modalities. Many alternative proof strategies can be implemented starting from the basic inference mechanisms offered by OTTER, as extensively reported in [CMPr90] and [CMPr91], allowing to tailor the proof style to the problem under consideration.

4 The V2O automatic semantic extractor

V2O is written in C for a UNIX environment, and is designed to work with the VTIP integrated platform for the analysis of VHDL produced by CLSI [CLSI90]. The overall semantic extraction environment is outlined in Fig. 2.

The translator program is divided in three functional blocks and relies on three libraries:

- the controller module interacts with the user and manages the calls to the procedures contained in the other two modules,
- the translator implements the semantic extraction criteria that perform the axiomatization of the single VHDL primitives and constructs, starting from the internal database format of the VTIP Design Library System (DLS), and yielding an intermediate format semantics stored in the Otter Library System (OLS).
- the Otter File Generator (OFG) is the only OTTER-dependent module, and produces the input file for the target verification system, then contained in the OFG library, from the intermediate form description of the OLS.

In the following paragraph, we detail the characteristics of the main modules of the translation system.

The VTIP platform. VTIP platform utilities allow to syntactically analize the source file, to create a database containing all the syntactic information, and to query the database. The platform is structured in a highly modular way: the lexical and syntactic analysis of the source file is done by the *VHDL analyzer*, the Design Library System contains the analyzed information in an internal format. To retrieve the DLS information,

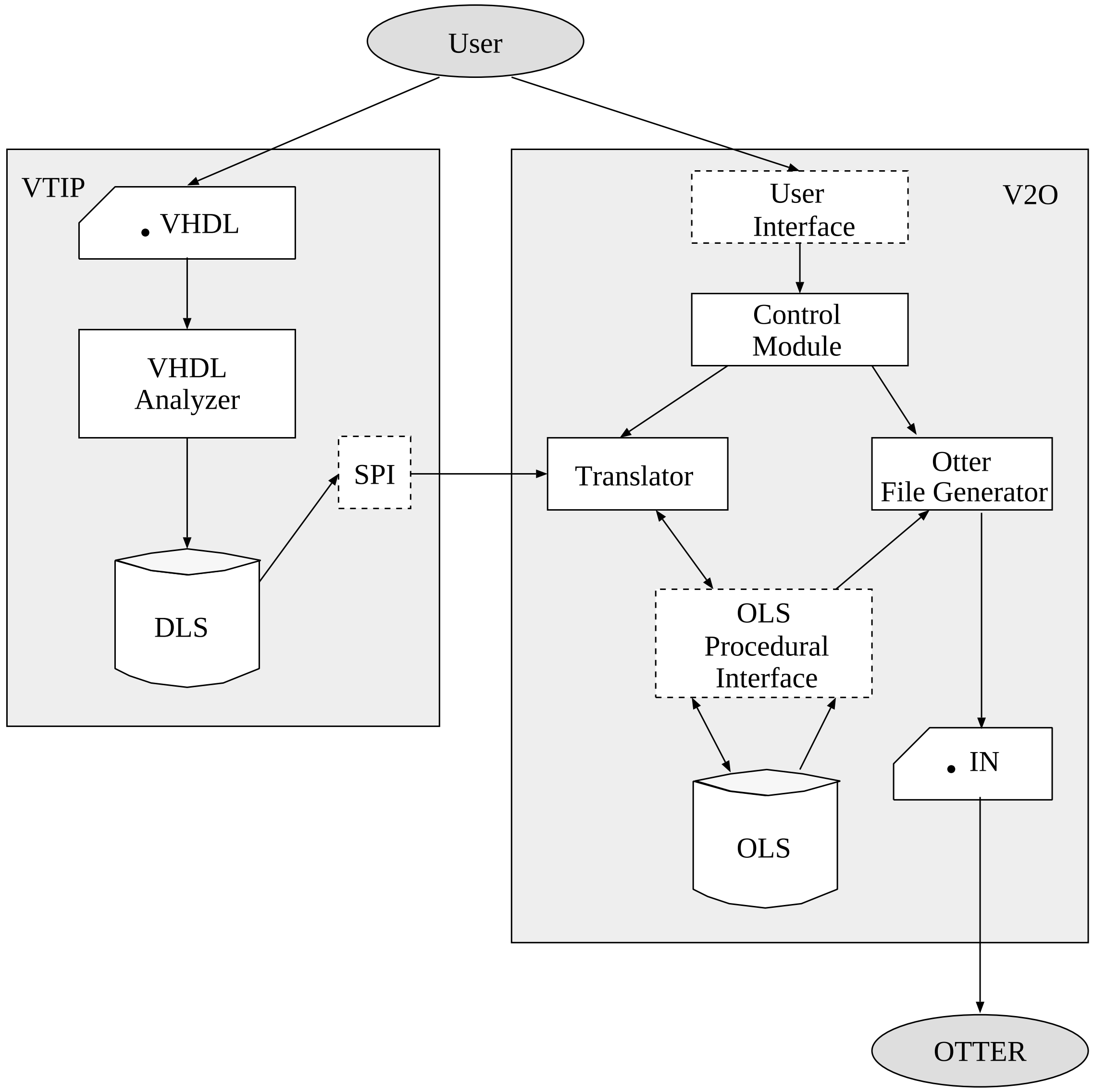

Figure 2: The semantic extraction environment

a *Software Procedural Interface* (SPI) is available. Its routines are system-independent C language procedures, thus highly portable, called at need by the translation program.

The DLS database is designed in an object-oriented approach: the backbone of the complex graph is a forest, where each tree contains the information relative to a description, and where tree nodes are instances of objects. The many node types (more than 150) are organized in classes, each class inheriting the attributes of the parents, and adding the own ones. Nodes correspond to 'concepts' in VHDL, while attributes may be either data, stored in the node, or properties, recorded via pointers to other graph nodes. Thus, main links connect nodes within a tree, and secondary ones represent properties.

The Controller. This is the command nucleus of the system. For each VHDL unit, the corresponding DLS description must be generated first: one for each architecture, plus one for the entity, since this might contain constant and variable declarations. Other VHDL architectures involved with the current one (e.g., subcomponents) must be added to complete the description.

A *worklist* of the units to be linked is maintained to avoid double inclusion of submodules in the set of components to be translated, Procedures BuildEntity, BuildArchitecture, BuildConfiguration, BuildPackage first check whether needed units are already in the worklist, then whether they are already in the library of translated modules. In this case, the final semantic formulæ are already available, and are directly linked in the OTTER file.

The worklist of the subunits determines the sequence in which the translation to intermediate form is performed. The rules for subunit inclusion are context sensitive: as an example, a component declaration can appear inside an Architecture, but not inside an Entity. The nesting of units is allowed, since the Build procedures are recursive. Each translated subunit corresponds then to a distinct element in the OLS library.

The Translator. As translation target, a VHDL unit is indicated. The module looks for the unit in the DLS, extracts the semantic information and translates individually each statement in a clause or rewrite rule in an OTTER-independent intermediate format. The result of this first interpretation step is stored in the Otter Library System OLS.

The intermediate format has a very simple grammar whose prefix notation. eases the second translation step towards the (mostly Lisp-like) input languages for theorem provers. Among the advantages of introducing this intermediate form, we can mention the ease of writing a parser for such a small language, the readability and compactness of the format, which allows a more efficient memory use, the possibility for the user to directly act on the objects, moving, deleting or modifying files. In addition to that, it is easy to retarget the final translation step for other verification systems that accept rewrite rules: this intermediate representation is at a higher level than the specific input format of a prover.

Library elements beginning with the *OT identifier contain semantics directly in OTTER format, ready for inclusion in the final file. This feature has proven very useful in the test phase, as well as for dealing with language primitives not yet automatically translated, for which the final OTTER semantic was known. As a drawback, it may also be a source of errors, due to the direct user intervention in the contents of the OTTER file.

The code of the module has been structured in many small procedures, so that it be easily retargetable, modifiable or extensible for new constructs of VHDL.

The primitives and constructs for which automatic support is already provided are:

- *Units*: entity, architecture, package declarations, package body, configuration specification,

- *concurrent instructions*: concurrent variable assignment, component instantiation, process, conditional assignment of signals, concurrent assert instruction

- *sequential instructions*: variable and signal assignment, case, subroutine calls, assert instruction, null instruction,
- *type declarations*: bit, integer, vector, enumerative types, subtypes of the previous types, record types,
- *Object declarations*: declarations of constants, variables, signals, procedures, functions, components .

The OTTER Library System. It includes the internal format semantics for each VHDL unit (module or component) already translated in the internal format. It is organized as a serie of directories, where each unit is stored in a file. This file contains the descripton of the single unit in intermediate format, created by the translator The translation to OTTER format is now straightforward.

The OLS Procedural Interface eases the access to the OLS system: procedures search for the referenced units in the OLS directory, returning the handle of the file, insert new units in intermediate format. A parsing subroutine extracts data from the OLS files, exchanged in the form of *structures*, each containing a rewrite rule of the specified OLS unit. A careful choice of the data structures has taken into account their versatility (there are parameterized types and functions), modularity and ease of extension.

The OTTER File Generator. This module assembles the input file for OTTER: under the supervision of the controller, it is responsible for the correct creation of the different sections of the file. First, the standard header file oheader.in is included. It contains OTTER commands and options that enforce and direct the proof strategy. It is now designed for rewrite-rule based proofs with final resolution step, but a set of headers could be easily made available, to choose among the several proof styles outlined in [CMPr91].

The OFG accesses the OLS for the desired VHDL units, and performs the second translation step generating the final OTTER format; the set of rewrite rules and definitions so produced are then written in the file. It links them to the necessary theorems, chooses the verification options and returns a file ready for being input to the automatic processing.

Before writing the final file, some optimizations are possible: operator definitions (and related theorems) not involved in the descriptions may not be linked, thus reducing the number of rules; moreover, options that slow down the execution (like demod_linear) are not always needed.

At the end of the execution, the OFG library contains the executable OTTER files.

The user interface. The interaction between user and program is completely under the supervision of the controller. The user must input the fundamental data: name of the entity, of the two architectures to be verified, and of the target OTTER file. The program then reconstructs the full names of the two architectures to be compared, and passes them to the SPI and the translator. After that, the translation follows automatically, as previously outlined.

The control is distributed in the different parts of the main program, since the procedures for the other blocks are called in several distinct points. The OFG and OLS libraries are opened, calls to high level procedures constitute the interface towards the controller.

In the actual prototypical implementation, the user still writes the goal clause in ther OTTER file, but the task is easily automatable for large categories of circuits. In fact, output signals of an entity can be identified automatically, so that the goal for combinational circuits can be generated directly by the translator.

For sequential circuits one has to know what kind of proof is better performed. The user must then precise the best proof methodology and indicate the complete name of a signal, common to both architectures, to be used as time reference (e.g., for Finite State Machines, the register clock).

For other kinds of equivalence checks, like *observational equivalence*, state signals must be accessed too, and a mapping function between corresponding states of different machines might be needed. One possible viable solution to automate its generation is by exploiting the dependency relations already available in the structural analysis of the descriptions, but the current implementation of the prototype does not include this feature.

The error diagnosis is partly linked to the one in VTIP, and partly relative to the management of the new libraries and files: missing, badly referenced or named units are detected, as well as illegal library operations.

5 Example and results

As a compact well known example of RT_level circuit with a data path and a control part we have taken the Traffic Light Controller of [LiSU89], where it is used as a comprehensive example to illustrate the use of VHDL throughout the design process. The example is a system which controls a traffic light at the intersection of a highway and a farm road. A high-level specification models the behaviour of the system, and is given at the functional level as an Algorithmic State Machine [Clar73]. State variables are still to be assigned. In this example, a single architectural body is created, with a corresponding entity to define the interface, and a test architecture, containing a set of criteria for checking the operation of the device at this level. The specification is then successively partitioned, with a definition of the interfaces between the submodules, and their interaction modalities. Partitions are then independently refined into an implementation. The control part is then refined into a PLA circuit representation. At each stage, the correctness can be also checked via simulation and test.

In Appendix we include the OTTER file generated by V2O starting from the VHDL description of the specification and implementation of the TLC given in [LiSU89].

Some results relative to the TLC and to other circuits, obtained on a SparcStation II, with OTTER 2.2 are reported in Tab. 1. The table includes an *Alu*, a *Known Length Sequence Recognizer* (a sequential circuit of known period), an *Unknown Length Sequence Recognizer* (it recognizes a simple language with a finite, but unknown number of symbol repetitions), a *BCD code recognizer*, and the 16-bit counter *C16bs* (already verified with

Circuit	ALU	KLSR	ULSR	BCD	C16bs	TLC
total CPU time (s)	0.32	0.23	0.29	0.22	3.97	28.31
system time	0.07	0.07	0.04	0.09	0.10	0.09
input time	0.15	0.13	0.09	0.08	0.23	0.59
hyper_res time	0.02	0.00	0.00	-	0.01	0.04
demod time	0.09	0.02	0.14	0.01	3.66	26.95
print_cl time	0.01	0.00	0.01	0.01	0.01	0.18
# of rewrites	1112	171	1577	69	52605	38555

Table 1: Results on the CHARME benchmarks

different strategies from an ART$*$ description, see [Marg92]). KLSR, ULSR, BCD, a counter and a TLC are benchmarks circuits of the ESPRIT BRA3216 'CHARME' project, chosen for an evaluation of several theorem provers. The selection criteria of those circuits was not by size, which is rather small, but privileged their belonging to different classes of sequential circuits. The consequent need of resorting to different verification strategies was expected to challenge the different provers under examination within the project.

Acknowledgements

This work has been partially supported by an ERASMUS scholarship at the Laboratoire Artemis, IMAG, Grenoble (France). The authors are grateful to Prof. Dominique Borrione and Dr. Ashraf Salem for their valuable contributions during several helpful discussions.

References

[BEFP89] A. Bratsch, H. Eveking, H.J. Faerber, M. Kelelatchew, J. Pinder, U. Schellin: "LOVERT - A logic verifier of register-transfer level descriptions," IMEC-IFIP Workshop on "Applied Formal Methods For Correct VLSI Design", Leuven (Belgium), November 1989, Vol. 2, pp. 522-531.

[BoPr89] D. Borrione, P. Prinetto: "Zero-defect designs, why and how: formal verification vs. automated synthesis," Information Processing 89, Elsevier Science Publishers, 1989, pp. 233-240.

[BoPS91] D. Borrione, L. Pierre, A. Salem: "PREVAIL: a Proof Environment for VHDL Descriptions," Proc. of the Advanced Research Workshop on "Correct Hardware Design Methodologies", Turin (Italy), June 1991

[Clar73] C. Clare: "Designing Logic Systems Using State Machines," McGraw-Hill, 1973.

[CLSI90] CAD Language Sysyem Inc.: "VHDL Tool Integration Platform (VTIP)", CAD Language Sysyem Inc., Maryland (USA), 1990

[CMPr90] P.Camurati, T.Margaria, P.Prinetto: "The OTTER environment for resolution-based proof of hardware correctness," Proc. Workshop on "Designing Correct Circuits," September 1990, Oxford (UK), Springer Verlag.

[CMPr91] P.Camurati, T.Margaria, P.Prinetto: "Formal verification of design correctness of sequential circuits based on theorem provers," IEEE CompEuro'91: Advanced Computer Technology, Reliable Systems and Applications, Bologna (Italy), May 1991, pp. 322-326.

[Evek85] H. Eveking: "The application of CHDL's to the abstract specification of hardware," CHDL '85: IFIP 7th Int. Symposium on Computer Hardware Description Languages and their Applications, Tokyo (Japan), August 1985, pp. 167-177.

[GrTe91] M. Griva, R. Tesio: "V2O User's manual," Technical report, Laboratoire Artemis, IMAG, Grenoble (France), 1991.

[IEEE88] IEEE Standard VHDL Language Reference Manual, IEEE Std. 1076-1987, IEEE, March 1988.

[Lang90] M. Langevin: "Automated RTL verification based on predicate calculus," CAV'90 - Workshop on "Computer-Aided Verification", DIMACS Series in Discrete Mathematics and Th. Computer Science, Vol.3, AMS, 1990, pp. 577-590.

[LiSU89] C. Lipsett, C.F. Schaefer, C. Ussery: "VHDL: Hardware description and design," Kluwer Ac. Publisher, 1989

[Marg91] T. Margaria: "First-Order theories for the verification of complex FSMs," Aachener Informatik-Berichte Nr.91-30, RWTH-Aachen, Aachen, December 1991.

[Marg92] T. Margaria: "Hierarchical mixed-mode verification of complex FSMs described at the RT level," IFIP TC 10/WG 10.2 Proc. Intern. Conference on "Theorem Provers in Circuit Design: Theory, Practice and Experience", Nijmegen (The Netherlands), June 1992, in IFIP Transactions A-10, North-Holland, pp 59-75.

[MaSt93] T. Margaria, B. Steffen: "Distinguishing Formulas for Free", EDAC–EUROASIC'93: IEEE European Design Automation Conference, Paris (France), February 1993.

[MaSt93b] "A System for Compositional Hardware Verification," Proc. Kolloquium on Programmiersprachen und Grundlagen der Programmierung, Techn. Report N. 9309, Fakultät für Informatik, Universität der Bundeswehr München, December 1993, pp. 203-216.

[McCu90] W.W. McCune: "OTTER 2.0 User's Guide," Rep. ANL-90/9 Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL (USA).

[PrGa85] P.Prinetto, S. Gai: "ART∗: user manual," Int. Rep. DAI/CAD n.14/85, Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino (Italy), October 1985.

A Translation procedures

Procedure TP translates VHDL processes. For each statement, it calls procedure TSS, that translates the single statements. As a convention, variables related to the VHDL description are in *emphasized* while the OTTER syntax is in sans serif.

```
procedure TP (in stat-list, sensitivity-list);

AS := {sensitivity-list };
CS := {};                   % signals in the conditional expression
CV := {};                   % variables in the conditional expression
JRV := {};                  % set of var. read by the preceding instructions
JAO := {};                  % set of objects modified by the prec. instructions
Fun[] := [];                % Fun[Obj] returns the stable value of Obj
Mem[] := [];                % Mem[Obj] = true if Obj has memory

TSS(stat-list, CS,CV,JRV,JAO,Fun[],Mem[],WObj);

for Obj in WObj do
    if Mem[Obj] then                     % if the object has memory
        Write(v(Obj , [x| xr]) = if (e(AS, [x | xr]),Fun[ Obj ], v( Obj , xr).]);
    else % the object has no memory
        Write(v( Obj , [x | xr]) = Fun[ Obj] ).);
    end if;
end for;
end procedure TP;
```

```
procedure TSS(in stat-list; in CS,CV; inout JRV,JAO, Fun[], Mem[]; out Wobj);

WObj := {};

for each statement in stat-list do
    case statement of

    null : begin end;

    assert : begin end;

    signal assignment S <= waveform: begin
        expr := last_el of waveform;     % extracts the stable value
        RS := ExtractS(expr) ∪ CS;       % read signals
        RV := ExtractV(expr) ∪ CV;       % read variables
        RAS := RS ∩ AS;                  % read sign. in the sensitivity list
        RNS := RS  AS;                   % read sign. not in the sensitivity list
        RPV := RV ∩ JAO;                 % read vars already assigned before this instr.
        RSV := RV  RPV;                  % read vars. not yet assigned
        JRV := JRV ∪ RV;                 % update just read variables
        JAO := JAO ∪ { S }               % set of just updated objects
```

```
        Fun[S] := expr (v(RAS,i ), v(RNS, i-1 ), v( RPV,i), v(RSV, i-1));
        Mem[S] := ¬((RNS = φ) ∧ (RSV = φ) ∧¬ Mem[RV]);
        WObj := WObj ∪ { S };
    end;

    variable assignment  V := expression: begin

        if V in JAO then Error(Reassigned variable) else

           ⋮                                    % set assignments as for the signal assignment

        JAO := JAO ∪ {V}          % set of just updated objects
        Fun[V] := expr (v( RAS,i), v( RNS, i-1 ), v( RPV,i ), v( RSV, i-1));
        Mem[V] := ¬((RNS = φ) ∧¬ (V ∈ JRV) ∧¬ Mem[RV]);
        WObj := WObj ∪ {V };
    end;

    if condition then stat1 : begin

        CS' := ExtractS(condition);
        CV' := ExtractV(condition);
        OldJAO := JAO;

        TSS(stat1, CS'∪ CS, CV' ∪ CV, JRV, JAO, Fun'[], Mem'[], WObj');

        for Obj in WObj do
           if Obj ∈ OldJAO then
              Fun[Obj] := if ( condition, Fun'[Obj], Fun[Obj] );
              Mem[Obj] := Mem'[Obj] ∨ Mem[Obj];
           else
              Fun[Obj] := if ( condition , Fun'[Obj] , v(Obj, i-1));
              Mem[Obj] := Mem'[Obj];
           end if;
        end for;
        Wobj := Wobj ∪ WObj';
    end;

        ⋮

    end case;
end for;
end procedure TSS;
```

B Traffic Light Controller

Here is the OTTER description generated by V2O:

```
list(demodulators).

% Rewrite rules describing the TLC circuit

% constants declarations
(_TlBehHg_state = [0, 0]).
(_TlBehHy_state = [0, 1]).
(_TlBehFg_state = [1, 0]).
(_TlBehFy_state = [1, 1]).

% function State_to_bits
(State_to_bits(xI) = ifi(eq(xI, int(0)), _TlBehHg_state,
                     ifi(eq(xI, int(1)), _TlBehHy_state,
                     ifi(eq(xI, int(2)), _TlBehFg_state,
                        _TlBehFy_state)))).

% function Bits_to_state
(Bits_to_state([x1, x2]) = ifi(x1, ifi(x2, int(3), int(2)),
                                   ifi(x2, int(1), int(0)))).

% architecture Beh of Tl_controller

(v(_TlBehY(z), [x|xr]) = _TlBehState_to_bits(v(_TlBehController_state(z), xs))).

(v(_TlBehSumofproducts(z), xs) =
 ifbv(and(not(v(_TlBehC(z), xs)), and(not(el(v(_TlBehY(z), xs), int(0))),
      not(el(v(_TlBehY(z), xs), int(1))))), [0, 0, 0, 0, 0, 1, 0],
 ifbv(and(not(v(_TlBehTl(z), xs)), and(not(el(v(_TlBehY(z), xs), int(0))),
      not(el(v(_TlBehY(z), xs), int(1))))), [0, 0, 0, 0, 0, 1, 0],
 ifbv(and(v(_TlBehC(z), xs), and(v(_TlBehTl(z), xs),
      and(not(el(v(_TlBehY(z), xs), int(0))),
      not(el(v(_TlBehY(z), xs), int(1)))))), [0, 1, 1, 0, 0, 1, 0],
 ifbv(and(not(v(_TlBehTs(z), xs)), and(not(el(v(_TlBehY(z), xs), int(0))),
      el(v(_TlBehY(z), xs), int(1)))), [0, 1, 0, 0, 1, 1, 0],
 ifbv(and(v(_TlBehTs(z), xs), and(not(el(v(_TlBehY(z), xs), int(0))),
      el(v(_TlBehY(z), xs), int(1)))), [1, 1, 1, 0, 1, 1, 0],
 ifbv(and(v(_TlBehC(z), xs), and(not(v(_TlBehTl(z), xs)),
      and(el(v(_TlBehY(z), xs), int(0)),
      el(v(_TlBehY(z), xs), int(1))))), [1, 1, 0, 1, 0, 0, 0],
 ifbv(and(not(v(_TlBehC(z), xs)), and(el(v(_TlBehY(z), xs), int(0)),
      el(v(_TlBehY(z), xs), int(1)))), [1, 0, 1, 1, 0, 0, 0],
```

```
 ifbv(and(v(_TlBehTl(z), xs), and(el(v(_TlBehY(z), xs), int(0)),
      el(v(_TlBehY(z), xs), int(1)))), [1, 0, 1, 1, 0, 0, 0],
 ifbv(and(not(v(_TlBehTs(z), xs)), and(el(v(_TlBehY(z), xs), int(0)),
      not(el(v(_TlBehY(z), xs), int(1))))), [1, 0, 0, 1, 0, 0, 1],
 0, 0, 1, 1, 0, 0, 1])))))))))).


(v(_TlBehController_state(z), xs) =
 Bits_to_state([el(v(_TlBehSumofproducts(z), xs), int(0)),
                el(v(_TlBehSumofproducts(z), xs), int(1))])).
(v(_TlBehSt(z), xs) = el(v(_TlBehSumofproducts(z), xs),
 int(2))).
(v(_TlBehHl(z), xs) = [el(v(_TlBehSumofproducts(z), xs),
 int(3)), el(v(_TlBehSumofproducts(z), xs), int(4))]).
(v(_TlBehFl(z), xs) = [el(v(_TlBehSumofproducts(z), xs),
 int(5)), el(v(_TlBehSumofproducts(z), xs), int(6))]).


% Initial values for input variables
(v(_TlBehC(Test), []) = if(C, 1, 0)).
(v(_TlBehTl(Test), []) = if(TL, 1, 0)).
(v(_TlBehTs(Test), []) = if(TS, 1, 0)).

% Initial values for input variables
(v(_TlBehC(Test), []) = if(C, 1, 0)).
(v(_TlBehTl(Test), []) = if(TL, 1, 0)).
(v(_TlBehTs(Test), []) = if(TS, 1, 0)).

% architecture Str of Tl_controller

(v(_TlStrYIn(z), x) = State_to_bits(v(_TlStrController_state(z), x))).
(v(_TlStrController_state(z), x) = Bits_to_state(v(_TlStrYout(z), x))).
(_PlaBeh1In0([Impl|z]) = _TlStrC(z)).
(_PlaBeh1In1([Impl|z]) = _TlStrTl(z)).
(_PlaBeh1In2([Impl|z]) = _TlStrTs(z)).
(_PlaBeh1In3([Impl|z]) = el(_TlStrYin(z), int(0))).
(_PlaBeh1In4([Impl|z]) = el(_TlStrYin(z), int(1))).
(_TlStrYout(z) = [_PlaBeh1Out0([Impl|z]), _PlaBeh1Out1([Impl|z])]).
(_TlStrSt(z) = _PlaBeh1Out2([Impl|z])).
(_TlStrHl(z) = [_PlaBeh1Out3([Impl|z]), _PlaBeh1Out4([Impl|z])]).
(_TlStrFl(z) = [_PlaBeh1Out5([Impl|z]), _PlaBeh1Out6([Impl|z])]).

% architecture Beh1 of Pla

(_PlaBeh1Plaouts(z) = [
  [0, 0, 0, 0, 0, 1, 0],
```

```
  [0, 0, 0, 0, 0, 1, 0],
  [0, 1, 1, 0, 0, 1, 0],
  [0, 1, 0, 0, 1, 1, 0],
  [1, 1, 1, 0, 1, 1, 0],
  [1, 1, 0, 1, 0, 0, 0],
  [1, 0, 1, 1, 0, 0, 0],
  [1, 0, 1, 1, 0, 0, 0],
  [1, 0, 0, 1, 0, 0, 1],
  [0, 0, 1, 1, 0, 0, 1]]).

(v(_PlaBeh1Newstate(z), xs) =
  ifi(and(not(v(_PlaBeh1In0(z), xs)), and(not(v(_PlaBeh1In3(z), xs)),
      not(v(_PlaBeh1In4(z), xs)))),int(0),
  ifi(and(not(v(_PlaBeh1In1(z), xs)), and(not(v(_PlaBeh1In3(z), xs)),
      not(v(_PlaBeh1In4(z), xs)))),int(1),
  ifi(and(v(_PlaBeh1In0(z), xs), and(v(_PlaBeh1In1(z), xs),
      and(not(v(_PlaBeh1In3(z), xs)), not(v(_PlaBeh1In4(z), xs))))),int(2),
  ifi(and(not(v(_PlaBeh1In2(z), xs)), and(not(v(_PlaBeh1In3(z), xs)),
      v(_PlaBeh1In4(z), xs))),int(3),
  ifi(and(v(_PlaBeh1In2(z), xs), and(not(v(_PlaBeh1In3(z), xs)),
      v(_PlaBeh1In4(z), xs))),int(4),
  ifi(and(v(_PlaBeh1In0(z), xs), and(not(v(_PlaBeh1In1(z), xs)),
      and(v(_PlaBeh1In3(z), xs), v(_PlaBeh1In4(z), xs)))), int(5),
  ifi(and(not(v(_PlaBeh1In0(z), xs)), and(not(v(_PlaBeh1In1(z), xs)),
      and(v(_PlaBeh1In3(z), xs), v(_PlaBeh1In4(z), xs)))), int(6),
  ifi(and(v(_PlaBeh1In1(z), xs), and(v(_PlaBeh1In3(z), xs),
      v(_PlaBeh1In4(z), xs))),int(7),
  ifi(and(not(v(_PlaBeh1In2(z), xs)), and(v(_PlaBeh1In3(z), xs),
      not(v(_PlaBeh1In4(z), xs)))),int(8),
  int(9)))))))))).


 (v(_PlaBeh1Out0(z), xs) =
  el(el2(_PlaBeh1Plaouts(z), v(_PlaBeh1Newstate(z), xs)), int(0))).
(v(_PlaBeh1Out1(z), xs) =
  el(el2(_PlaBeh1Plaouts(z), v(_PlaBeh1Newstate(z), xs)), int(1))).
(v(_PlaBeh1Out2(z), xs) =
  el(el2(_PlaBeh1Plaouts(z), v(_PlaBeh1Newstate(z), xs)), int(2))).
(v(_PlaBeh1Out3(z), xs) =
  el(el2(_PlaBeh1Plaouts(z), v(_PlaBeh1Newstate(z), xs)), int(3))).
(v(_PlaBeh1Out4(z), xs) =
  el(el2(_PlaBeh1Plaouts(z), v(_PlaBeh1Newstate(z), xs)), int(4))).
(v(_PlaBeh1Out5(z), xs) =
  el(el2(_PlaBeh1Plaouts(z), v(_PlaBeh1Newstate(z), xs)), int(5))).
(v(_PlaBeh1Out6(z), xs) =
  el(el2(_PlaBeh1Plaouts(z), v(_PlaBeh1Newstate(z), xs)), int(6))).
```

```
% Initial values for memory variables
(v(_TlStrYController_state(z), []) = if(STATE, 1, 0)).
(v(_TlStrYin(z), []) = [if(Y0, 1, 0), if(Y1, 1, 0)]).

% Initial values for input variables
(v(_TlStrC(Test), []) = if(C, 1, 0)).
(v(_TlStrTl(Test), []) = if(TL, 1, 0)).
(v(_TlStrTs(Test), []) = if(TS, 1, 0)).

(v([xe|xr], x) = [v(xe, x)|v(xr, x)]).
(v([], x) = []).
(v(el(x, xi), xs) = el(v(x, xs), xi)).

% Oracle List
(OL1 = [Bit0]).
%(OL2 = [Bit1, Bit0]).

end_of_list.

list(sos).
(v(_TlStrHl(Test), []) != v(_TlBehHl(Test), []))
| (v(_TlStrFl(Test), []) != v(_TlBehFl(Test), []))
| (v(_TlStrSt(Test), []) != v(_TlBehSt(Test), []))
| (v(_TlStrController_state(Test), []) != v(_TlBehController_state(Test), [])).

end_of_list.
```