First-Order theories for the verification of complex FSMs

Tiziana MARGARIA*

Dipartimento di Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi 24
I-10129 Torino TO
Italy

Aachener Informatikberichte
Nr. 91 – 30

Abstract

A formal framework for efficient verification of Finite State Machines with data path and control unit is presented. A homogeneous incremental approach to the definition of gate level and Register Transfer level semantics allows to build a unique compositional semantics that encompasses both design levels. In particular, criteria for axiomatizing Register Transfer behavioural and structural descriptions in the ART$*$ language allow to express their intended functionality within a uniform *semantic framework* based on First-Order Logic, in a *formal verification system*. Proving the correctness of a circuit with respect to its specifications is then amenable to a proof of logic equivalence between the respective semantic formulæ. Within this framework, the proposed verification methodology allows efficient hierarchical, mixed-mode verifications. Proofs are performed automatically by means of the OTTER general-purpose theorem prover. Experimental results show a performance improvement of a factor 10 over the known RT-level verification techniques.

Keywords: RT level verification, Semantics, First-Order Theories, Finite State Machines, Theorem Proving, OTTER

*This work has been supported by the Graduiertenkolleg at the RWTH Aachen.

1 Introduction

The formal verification of the design of "*complex*" Finite State Machines (FSM's), i.e., machines having both a control unit and a data path that in each state performs a set of nontrivial operations, is still an open problem. Methods have been proposed for separately verifying the data path [NKFT90] and the control unit [CMPr91], while known integrated approaches either internally retain a distinction between control programs and data path graphs [Vemu89], or their scope is limited to kernels of Register Transfer level languages [Evek85], [BEFP89], [Lang90].

Aiming at providing a completely integrated and structured verification methodology for real life designs at the Register Transfer level, circuit descriptions are required to combine well defined semantics with the full power of a RT level Hardware Description Language.

We develop a verification framework based on semantic definitions in terms of First-Order theories, which captures both gate-level and RT-level descriptions. The proposed verification methodology allows us to prove the correctness of a circuit with respect to its specifications by establishing logical equivalence between their respective semantics, which are both given in terms of First-Order formulæ. The strength of the proposed verification method lies in its compositionality, which is implied by the incremental definition of the RT level semantics as an extension of the gate level semantics, and the formalization of the ART$*$ primitives in terms of RT level basic functions. Moreover, the use of a common semantic domain enables mixed-mode verification. Proofs of correctness can be carried out automatically and entirely at the semantic level, by means of the OTTER general-purpose First-Order Logic theorem prover [McCu90].

The framework is illustrated using ART$*$ [PrGa85], a hierarchical, multilevel HDL, with a rich set of primitive constructs and well defined semantics for the hardware description.

The paper is organized as follows: Section 2 introduces the design proce ss with special attention to the Register Transfer level and relative description techniques, Section 3 introduces the formal verificati on framework, relating circuit models with the corresponding semantic First-Order theories, and the constructive method to obtain a compositional semantics for gate and RT abstraction levels. Section 4 presents the proposed compositional verification methodology, and the verification tool, the OTTER general-purpose theorem prover for First-Order Logic. Section 5 informally describes the semantic concepts on which the axiomatization of a specific RT level language, ART$*$, is based, then an application is discussed in Section 6, with comparative re sults obtained with different verification methods.

2 Hardware description and design levels

The Register Transfer level identifies a precise design phase in the development of a hardware system, and it admits a rich set of possible description techniques, that still enable a formalization at several abstraction levels.

2.1 Design

In a top-down design, it is quite natural to distinguish at least three different design levels; each may be seen as a *specification* for the following one, in which the chosen design style determines the introduction of refinements and more detailed information.

- *High level specifications* express algorithmic requirements in terms of a sequence of computation steps, and are thus still implementation independent.

  As an intuitive example, the specification of a multiplier on natural numbers representable on 8 bits may be formulated at this level by requiring that both operands must be smaller or equal to 255, and that the circuit must perform their multiplication.

- *Architectural partitioning* into a set of modules allows to choose for each module a specific design style (e.g., microprocessor-based or special-purpose-machine implementation).

  In the case of the multiplier, a possible partitioning leaves the test on the input data to a comparator module, while the operation execution will require registers, and an adder or an ALU. A possible alternative would be the implementation of the multiplier as systolic array.

- At the *Register Transfer* design level, hardware modules are often detailed in terms of a *complex FSM*. The design process aims at (manually or automatically) synthesizing both the simple FSM corresponding to the control unit and the data path components, via *operation scheduling* and *resource allocation* steps. The resulting implementation is a circuit whose basic building blocks are elementary hardware functions, such as MSI/LSI components[1] in board oriented designs, or cells in ASIC-oriented designs[2].

  For the multiplier, once the implementation as complex FSM is chosen, the selected multiplication strategy (serial-parallel, parallel-parallel) determines the subsequent design of the data path (how many registers, simple or shift registers, choice between 8-bit ALU and an adder), and of the control unit. The tradeoff between the complication of the circuit and computation speed is such that the "best" scheduling and allocation for this single module is the one that optimizes the performance of the global system.

- At the *gate* level, the *simple FSM* corresponding to the control unit is designed starting from the RT-level description, and implemented in all details in terms of gates and flip-flops (SSI chips)[1]. The data path is usually made of RT-level standard components, therefore the design is detailed at the gate level only in case of ASIC-oriented design.

  For the multiplier, in a traditional implementation the multiplication strategy determines the subsequent gate-level design of the control unit: the minimal number of flip-flops is a consequence of the number of states, but it is possible to choose between minimal and redundant state encodings, and among different kinds of flip-flops.

2.2 Descriptions

Among the possible description techniques for complex FSMs at the RT design level, one of the most intuitive and commonly used by designers is based on Algorithmic State Machines [Glus68], [Clar73], where a synchronous sequential machine, the *control automaton*, cooperates with another machine, the *operational automaton*, to perform the data processing. In the hardware domain, they model the *control unit* and the *data path* activities, respectively. The control unit is in charge of correctly sequencing the data path by sending *commands* and reacting according to the values of returned *status signals*.

In such a model, the following description levels can be distinguished (see examples in Section 6):

- at the *functional* level the timing of the operations has been *scheduled*, hence the number of states is known. For each state, the state transition and output functions of the control unit and the operations to be executed concurrently by the data path components are precised.

  The structure of a functional description entry is

  $$< state > \; : \; < condition\_expression > \longrightarrow < data\_path\_op\_set > \; ; \; < next\_state >$$

  where the *condition_expression* constitutes the precondition for selecting one among the sets of concurrent *data path operations* awaiting execution in the current *state*. A necessary restriction on the preconditions is that in each state they must be mutually exclusive (nondeterminism in the choice of the operation set is thus excluded), and that the data path operations in each set must be hardware compatible (e.g., multiple assignments to the same destination resource are not allowed).

  In addition, the *next state* is univocally determined by the control automa ton on the basis of its state transition function;

- at the *structural* level the machine is described in terms of encoded state variables and connections of structural building blocks. Typical blocks include counters, memory banks, shift registers, buffers, ALU's, PLA's, buses, together with additional logic gates, flip-flops, and latches to implement the glue logic.

  The description structure is similar to the one reported for the functional level, but state variables are now encoded, each signal's polarity is taken into account for correct connections, and each mutually exclusive operation set must prescribe an operation for each data path component. On the other hand, the structural RT level still abstracts from topological det ails of the basic building blocks and from their detailed timing requirements, such as clocking scheme or inertial delay.

[1] SSI, MSI and LSI stand for Small, Medium, and Large Scale Integrated circuits, respectively.
[2] ASIC stands for Application Specific Integrated Circuits.

3 The framework

The choice of a specific circuit *model*, i.e., the set of parameters that are considered sufficient to characterize the circuit in a specific design phase, determines the semantic level of abstraction. To follow a design from the high level specifications downto the circuital realization, at least six completely different circuit models are involved (Fig. 1), each of them inducing its own semantics. As a consequence, relating descriptions at different abstraction levels requires to bridge the semantic gap between the underlying models. This is in general a complex task, because abstraction and mapping functions are often awkward to derive and to handle, besides, they involve the need of relating both the *models*, and the *functional primitives* in charge of expressing the behaviour of the same circuit at the different levels.

A proof of correctness of a circuit requires the comparison between its specification and an implementation, which in general boils down to comparing objects of two different semantic domains.

- A direct approach requires to establish a *heterogeneous* specification-implementation relation linking different semantic domains, and therefore faces the complex task mentioned above.
- This could be avoided, if it were possible to find a *common semantic domain* for both the specifications and the implementations, which allows us to use a *homogeneous* specification-implementation relation.

In the next paragraph, we develop such a homogeneous framework, which captures the RT and the gate level.

3.1 Unifying the RT and the Gate Level

The RT and the gate level have very similar underlying *circuit models*. This allows us to provide these levels with a uniform *semantic model*. In more detail,

Design levels	Objects	Models
Algorithmic	system modules	integers natural numbers boolean expressions
Register Transfer	functional blocks	boolean expressions on bits and bitrectors
Gate	single gates, flipflaps	boolean algebra
Switch	single switches	multivalued logic
Transistor	electrical compponents	electrical parameters (R, C, T, V, I, P)
Layout	rectangles	design rules

Figure 1: Design levels and underlying semantic models

- on both levels HDL representations deal with *boolean operators* applied to *boolean constants* and *variables*, and their *present* and *past* values. This set of basic gate-level concepts, together with a *substitution mechanism* for replacing equals for equals, enables the representation of values on carriers and the introduction of a parameter passing mechanism that allows us to model wired connections between structural components, be they single gates or large functional blocks,
- at the RT level, one additionally reasons about *collections* of gates and flip-flops, rather than just single ones. However, it is still possible to retain the same basic gate level concepts here, and to consider the RT level-specific concepts as being derived from them in a straightforward way. To do so, we simply need to *expand* the set of primary operators, including operators on lists and more sophisticated control flow primitives, into gate level primitives. This gives us a uniform language that captures both the RT and the gate level.

Proving that a gate level implementation is correct with respect to a RT level specification therefore involves only one semantic model. In our framework, where the semantics is given in First-Order Logic, the verification is in fact reduced to proving either *logical implication* or *equivalence* of the First-Order formulæ, depending on the relative abstraction levels between the semantics of their HDL formalizations.

3.2 Semantics in terms of First-Order theories

The semantics of hardware descriptions is given in terms of First-order theories, i.e. in terms of axiomatizations in First-Order Logic with equality, together with some description-specific non-logical axioms expressing the semantics of hardware specific function and predicate symbols[3]. The non-logical axioms of the theory associated with hardware descriptions can be classified in two distinct groups:

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

- *HDL-specific* axioms, that reflect the semantics of the adopted language,
- *description-specific* axioms, that capture the semantics of each hardware description, determined by the activity invocations that constitute the description body.

Thus the semantics of hardware descriptions is essentially an axiomatization of First-Order Logic together with an *equational* characterization of all the HDL language constructs and some additional circuit dependent equations.

Following this intuition, the construction of the "theory" for a given hardware description requires the equational characterization of *gate level semantics*, *RT-level semantics* and *HDL-specific semantics*, which can be achieved once and for ever for each HDL, as well as the (automatic) translation of the (textual) hardware descriptions into equations.

Starting with First-Order Logic with equality, the axiomatization for a given hardware description is derived by means of the following four step methodology (see Fig. 2):

1. to capture the gate level model, we define the semantics on the basis of three fundamental concepts: a *boolean algebra*, a *time model*, and a *substitution mechanism*. The First-Order Logic kernel already present in the prover must be enriched with primitives that allow to substitute equals for equals in the circuit equations, thus transforming combinational subnetworks into boolean expressions, and to access past values of signals, so as to be able to model sequential circuits,
2. to ensure direct compatibility between the semantics defined for gate and RT level, great effort is spent in trying to build the second as an extension and refinement of the first one. The gate level objects, primitives, and laws are therefore simply included as kernel, and RT-level specific concepts are incrementally introduced to yield a fully compositional semantics encompassing both levels.

   More specifically, boolean variables and bit vectors are the only RT level types, so the only additional fundamental operators are the *lists selector* and *costructor*, plus a *conditional* primitive to express control statements.
3. to derive the *HDL-specific theory* associated with an RT level language, the following methodology is used:
   - a First-Order Logic formula is associated to each language primitive, to express its intended semantics in terms of the previously introduced semantic primitives, thus expanding the set of *axioms* of the theory,
   - a set of simplification rules and basic equivalences or useful reorderings are individuated, to become candidate *theorems* in the semantic theory,
   - those theorems are shown consistent with the axioms by proof with OTTER,
4. to give an axiomatization for each pair of descriptions, the procedure to be followed is determined by the syntactic structure of the specific descriptions. This is discussed in detail for ART∗ in Section 5.3.

The next Section introduces the verification methodology induced by the chosen axiomatization process, while Section 5 develops the theory-based semantics for the HDL ART∗.

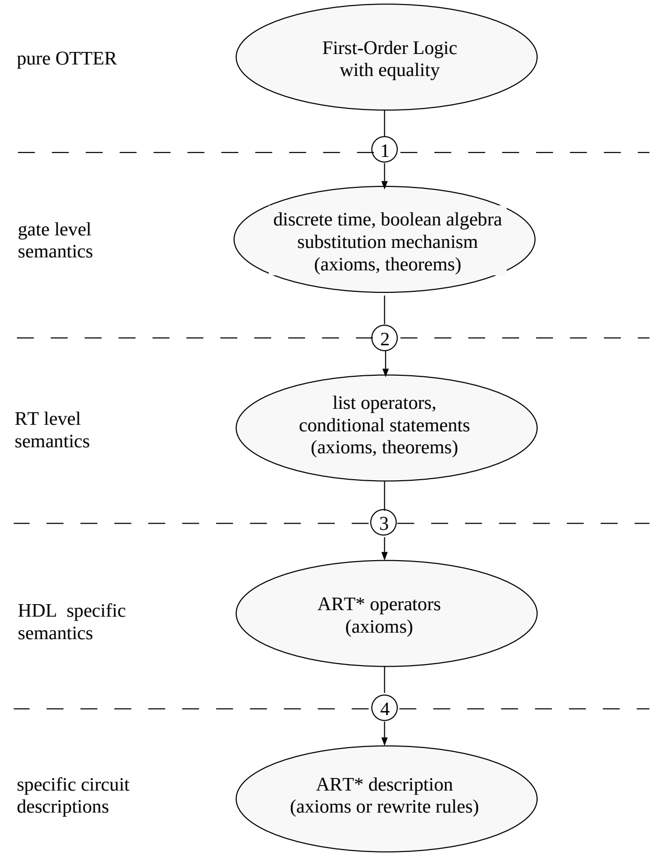

Figure 2: Incremental construction of the formal semantics for RT-level proofs.

4 The verification methodology

Our framework is based on the description of each individual HDL construct in terms of a First-Order theory, to yield a full axiomatization of the HDL, together with a set of simplification and canonicalization theorems. Each verification problem must then be properly formulated within this framework.

In order to verify the *functional equivalence* between a design and its specification, we first construct a characterizing First-Order theory for each of them (semantic extraction phase), and subsequently prove their *logical equivalence* (see. Fig. 3). 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 4.1).

The use of a *common semantic domain*, identified in the First-Order theories, reduces any *mixed mode* verification (i.e., verifications that involve different abstraction levels) to a homogeneous problem, where the compatibility between different descriptions is always ensured. Moreover, single modules or subsystems may be verified independently, so that the proof of their correctness can be separated from the check of the system's global correctness: this feature enables a *modular* and *hierarchical* verification methodology.

As a consequence, the suggested verification strategy optimally exploits the advantages offered by a top-down, hierarchical design methodology: according to it, complex circuits are hierarchically decomposed according to a functional partitioning strategy, and their design is refined in successive steps towards the desired implementation. In more detail:

1. functional partitioning into several blocks with different behaviour leads to a *hierarchy* of proofs:
   (a) when a design step is performed on any of the circuit's functional modules, the correctness of its new (partial) implementation w.r.t. its specification must be proved first,
   (b) since the overall behaviour must still comply with the global specifications, a subsequent global verification step is required,
2. the presence of subsystems with regular structure introduces *modularity* in their validation strategy:
   (a) first the correctness of the single cell must be shown,
   (b) then the correctness of the connections between cascaded or spatially iterated instanciations of the same correct cell is verified,
3. within the verification system, any unnecessary flattening at the semantic level should be avoided by resorting to *mixed mode* verification, in order to limit the overall complexity of the semantic formulæ .

If several correct semantic expressions for the same module are available, it is preferable, in practice, to include the one at the highest abstraction level. Once the First-Order formulæ are obtained, the reasoning is performed automatically by the theorem prover.

4.1 Using OTTER

OTTER is a resolution-based theorem prover with equality for First-Order Logic developed at the Argonne National Laboratories, IL (USA) [McCu90]. It is compactly coded in C (the executable

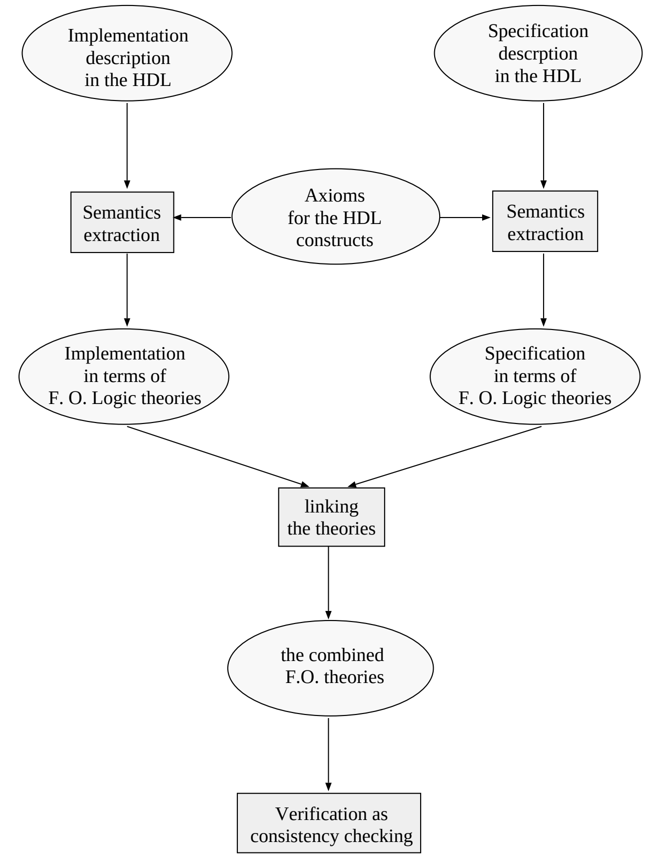

Figure 3: The verification process

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*. As a last-generation product, it is also capable of factoring, weighting, and term ordering.

Proofs in OTTER 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 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 [CMPr91], allowing to tailor the proof style to the problem under consideration.

The internal representation of any formula is in ITE (if-then-else) form, in a Binary Decision Diagram-like structure [BRBr90]. Weights are assigned to functions and variables, and a lexical ordering is introduced, in order to direct the prover in its choice of the best rewriting strategy. In addition, a list of goals may include some intermediate equalities to be proved as subgoals, which may then be reused in the course of the main proof. This increases the modularity of proof and optimizes the overall efficiency.

Section 6 demonstrates the impact of the choice of underlying verification and proof strategy, and the use of OTTER.

5 Application: the axiomatization of ART∗

ART∗ is a Register Transfer level, nonprocedural HDL, that allows behavioural, functional, structural, and multi-level descriptions, for both asynchronous and synchronous hierarchical design styles, derived from RTS 1a [Knob81]. It was developed at the Politecnico di Torino, as part of a RT level specification and simulation system, which is in extensive use since 1985. Kernel primitives (see Tab. 1) are direct extensions of notations used in switching theory. They are therefore easily and naturally amenable to a set of boolean expressions relating input and state variables. The importance of this feature is evident from the consequent simplification of the process of axiomatizing the language: ART∗ descriptions can map directly into real hardware structures, and the extraction of their semantics in terms of First-order theories is an easily automatizable task.

5.1 Language features

The time model The way time is represented distinguishes between models: ART∗ models a *discrete sampled time* rather than *streams*. This time model is appropriate for asynchronous as well as to synchronous design styles: the time flow is rendered with a *before/after* relation.

In OTTER, references to past values on the time scale are performed by means of finite difference equations: if t is the current time, $x(t)$ is the current value of signal x, [Pail86] introduces the accessor to previous values of a signal the *past functional* P:

$$P^i(x) \doteq x(t-i) \quad \forall i > 0.$$

Functions	Examples
constants, immediates	Vcc, Gnd, Float, #0B16, #37D8
carrier declaration	CK, A, F[0:3], MAR[0:12], MEM[0:255;0:11]
indexing	F[1:2], MAR[12:10], MEM[3], MEM[3;5]
addressing	MAR[A], MEM[MAR]
concatenation (row)	(F[3:0],A, #0B4)
concatenation (col)	(F[3:0]; #0B4; MAR[8:11])
boolean functions	AND, OR, NOT, EXOR, EXAND, NAND, NOR
arithmetic functions	ADD, ADC, SUB, SBB, INC, DEC
shift/rotate functions	ROR, ROL, RSH, LSH
compare functions	EQ, NEQ, GT, LT, GE, LE
fan_out functions	F(#0B1, 16)

Table 1: ART∗ basic primitives

Carriers In general, the behavior of hardware is represented by means of "*carriers*". In ART∗ *registers* model storage elements, while *terminals* model wired connections. The domain of observable values on any carrier at the RT-level is defined by the type *bool*, together with the usual set of associate predicates and functions.

In OTTER, ART∗ data types are mapped into the following structures:

- *scalar* carriers become (implicitly) universally quantified boolean variables
- *bit_vectors* are modelled as *lists of booleans*,
- *matrices of booleans* are represented as a vector of rows, and accordingly modelled as a *list of lists of booleans*.

Any proof of a theorem within the semantic theory is therefore conducted by *well-founded induction* within the theory of lists.

Activities A hardware description expresses the operational behaviour of a device by means of activity invocations. For the purpose of axiomatization, they define logical relationships between observations on carriers. In order to introduce the carriers' semantics, let us consider two simple cases of assignment described in ART∗, and their associate semantics:

- *Register "transfer" activity*: `x <= y` $\models$ $\forall t: \; x(t) = y(t-1)$.
  The corresponding semantics is expressed in OTTER by x = P(y),
- *Terminal "connect" activity*: `a = y` $\models$ $\forall t: \; a(t) = y(t)$,
  which in OTTER retains the same syntactic formulation as in ART∗.

As easily seen, while the clarity of definition is the same in the OTTER formalism and in predicate logic, the former does not need the explicit introduction of the time variable in the equations, which improves the readability of the formulæ.

Boolean algebra It is defined in terms of the postulates of switching theory [McCl86]. Since the OTTER implementation of First-Order Logic is not typed, it is necessary to state the domain for boolean variables explicitly. The basic knowledge concerning elementary gates (in terms of

truth tables) expresses the semantics of the *not*, and 2-input *and* and *or* gates. Since those equalities constitute the postulates of the theory, their correctness is assumed. All new relations must be proved correct with respect to them, before we can use them during the proofs as simplification and rewrite rules, in order to ensure consistency. The new goal is therefore to prove the most relevant theorems of switching theory, such as the existence of the null element, idempotence, involution, absorption, consensus, associative and commutative properties, De Morgan's laws, etc., thus gradually expanding the theory with consequences derived from the original kernel.

The chosen notation style is standard prefix functional form.

List operators To deal with registers, memories, bus structures and PLA's, all modelled in terms of lists, the basic list constructors, destructors and accessors are needed as additional primitive concepts.

This kernel is already present in OTTER, which allows both a LISP-like and a Prolog-like notation. The second one is chosen for compactness, ease of readability, and similarity to the notation of RT-level HDLs.

Control statements They constitute the kernel of any RT level description, and allow to neatly separate a hierarchically structured control flow from data processing activities.

As an example, let us consider two cases of conditional transfer activities:

- *transfer conditioned by terminal* `a`:
  `-IF- a -THEN- x <= y -FI-` $\models$
  $\forall t: \; (x(t) = z) \Leftrightarrow ((a(t) = 1) \wedge (z = y(t-1))) \vee ((a(t) = 0) \wedge (z = x(t-1)))$,
  corresponds to
  if (a, P(y), P(x)),
- *transfer conditioned by register* `w`:
  `-IF- w -THEN- x <= y -FI-` $\models$
  $\forall t: \; (x(t) = z) \Leftrightarrow ((w(t-1) = 1) \wedge (z = y(t-1))) \vee ((w(t-1) = 0) \wedge (z = x(t-1)))$,
  corresponds to
  if (P(w), P(y), P(x)).

Conditional connect activities are defined in a similar way, and `-CASE-` statements are transformed into nested conditional activities.

Substitution mechanism In a modular design, calls to previously defined description units may contain as actual parameters as well constants, as variables, as expressions, while the body of the unit itself refers to formal parameters as variables. To enable a correct parameter passing, a substitution mechanism capable of replacing equals for equals is needed. *Demodulation* suits the purpose, and it is already present in OTTER.

Composition rules The semantic theory embodies, as proved theorems, some composition rules like:

- functional algebra combination rules as presented in [Back78], which concern function composition, conditional forms, etc.
- specific rules for boolean and arithmetic modulo n algebras;

- simplification rules that allow to merge selection and concatenation functions on bitvectors,
- rules for the P functional [Pail86], like

$$\begin{array}{ll} Pf(x_1, x_2, \ldots, x_n) \doteq f(Px_1, Px_2, \ldots, Px_n) & \text{if } f \text{ is a function} \\ P\alpha \doteq \alpha & \text{if } \alpha \text{ is a constant.} \end{array}$$

All function compositions are given in *curried* form, and no quantification over functions is introduced. Since function evaluation is performed according to the *left-most inner-most* strategy, the operands of the evaluation are always values.

5.2 RT-level ART∗ specific concepts

5.2.1 Data structures

The structural elements of a system at this abstraction level are identified as *registers, terminals, memories, buses*, and *PLA* structures. In addition to the fundamental data structures already introduced,

- *buses* are seen as lists of the connected resources, and represented as a pair

$$< bus\_width, connection\_list >,$$

  with each connection characterized as $< bus\_bits, source, enable >$, while

- *PLA* (Programmable Logic Array) structures are represented as a list of input and output connections plus a pair of matrices (*and* and *or* plane respectively) in an espresso-like form.

5.2.2 Operands

Operands, corresponding to literals in an expression, can either be simple, like a terminal or a register, or *indexed, addressed, concatenated* or *complemented*. Combinations of the possibilities above often occur in hardware descriptions. For each of them, an informal definition is given here, together with the set of corresponding semantic primitives and their types, while the body of the definition in OTTER's syntax is reported in Appendix.

Indexing It is amenable to performing the appropriate *selection of a subset of elements* of the original (list-based) data structure. The following semantic functions have been defined:

- $sel(vector, index) : (bitv, int) \longrightarrow bit$
  extracts a single element from a bit_vector
- $sel\_vec(vec, el, range) : (bitv, int, int) \longrightarrow bitv$
  extracts a subvector of length $< range >$ starting from element $< el >$ of vector $< vec >$
- $sel\_mat(mat, row, row\_range, col, col\_range) : (boolmat, int^4) \longrightarrow boolmat$
  deals with submatrices extraction.
- selection of a single element (row) from a matrix is achieved by applying twice (once) the *sel_vec* function to the matrix.

Addressing Addressed operands are similar to indexed ones, but with a variable, rather than a constant value, in the selector. Conceptually, while indexing is associated with the idea of wiring, addressing is more akeen to multiplexing.
At the semantic level, the same selector functions are invoked. Since the *range* parameters are now variables, the evaluation cannot be carried out further and their expression is retained.

Concatenation It allows the user to group together operands for generating either a vector or a matrix, under the restriction that the dimensions of the elements be compatible. There are two basic ART∗ operators:

- The *comma* for concatenation in vector size, `[` $< op1 >$ `,` $< op2 >$`]`, concatenates along the rows:

$$(r \times c_1, r \times c_2) \longrightarrow (r \times (c_1 + c_2)).$$

  The operands can be of vector or scalar type.
- The *semicolon* for concatenation in matrix size, `[`$< matrix >$ `;` $< matrix >$`]`, concatenates vectors or matrices along their columns:

$$(r_1 \times c, r_2 \times c) \longrightarrow ((r_1 + r_2) \times c).$$

The corresponding semantic functions are a set of list *append* functions:

- $append(vec1, vec2) : (n, m) \longrightarrow n + m$
  is the usual append function; it concatenates scalars and/or (row) vectors in vector size
- $append\_vm(x, y) : ((n, m), (n, m')) \longrightarrow (n, m + m')$
  concatenates (column) vectors or matrices in vector size
- $append(mat1, mat2) : ((n, m), (n', m) \longrightarrow (n + n', m)$
  concatenates the rows in matrix size
- to catenate vectors and matrices in matrix size, the same *append* function above applies, but an (implicitely row) vector must be first transformed in a column vector via the function $to\_col(vec)$: $(r, 1) \longrightarrow (1, r)$.

Multiple concatenations reduce to repeated nested applications of the corresponding same *append* and *append_vm* functions.

Complementing It amounts to boolean complementing of each element of the operand.

5.2.3 Boolean expressions

ART∗ boolean operators can be grouped into three different sets:

- binary operators including *and, exand, exor* and *or*, are appl icable to all operands types (scalars, vectors, matrices) providing that dimensions are compatible. The application is bit-wise, to corresponding elements of the operands, and their tail-recursive definition immediately derives from a combination of the basic boolean algebra and list operators,
- unary operators are the same as above, but applied to vectors or matrices, to yield a scalar or vector, respectively. Several alternative choices are possible for the recursive definition of the semantic function, with a tradeoff between generality of the form and efficiency of computation. Efficiency is stressed here, since it better reflects the ART∗ informal semantics,

- the only two functions that may operate on an arbitrary number of operands are *nand* and *nor*.

Since nested expressions are allowed, the evaluation depends on implicit operator priority or explicit bracketing. The extracted semantics must reflect the proper evaluation sequence.
The semantics of multilevel combinational sub-networks is still compactly given by a single expression with parenthesized subexpressions. Internal carriers need not be declared as additional terminals.

5.2.4 Functions

The rich set of functions allows an easy description of standard components of combinational sub-networks. They can be grouped in *arithmetic*, *shift* and *compare* functions, and some special functions like the *fan_out* and *transpose*.

Arithmetic functions ART$*$ supports the full set of addition/subtraction operations, whose semantic expressions match the algorithm behind the ART$*$ function, which closely follow the circuital realization.

As an example, the semantic function of the *adc* (add-to-carry) operation reflects the hardware implementation as cascaded composition of full adders.

Semantic functions are provided for data representations with most significant bit right or left, the reason lying in tail recursion being the preferable structure for the semantic functions, while the conventional representation on bitvectors is with most significant bit at the left. The list $reverse(vec)$ function allows to match the two alternative representations.

A list of proved theorems about the functions can be used as simplification rules.

Shift functions The rotate and arithmetic shift functions are all supported in ART$*$. Each can take alternative forms, depending on whether the rotate/shift of one or more positions is required. The semantic expressions are given by nested list operators, like sublist selection and composition, plus the *fan-out* function for arithmetic shifts.

A set of theorems concerning partial or total cancellation of nested rotate/shifts is available, but their application sometimes requires the validity of additional side conditions.

Compare functions ART$*$ provides the full range of functions, of which EQ and NE take any operand, while GT, LT, GE and LE operate only on vectors interpreted as bit strings.
As for their semantics, the *eq* function can be applied both to values and to expressions, in the latter case by syntactic match of the expressions, rather than by arithmetic evaluation. The other functions reduce to the evaluation of arithmetic operations on the arguments, returning the value of the carry bit.

5.2.5 Submodules

Several instances of one description template can be embedded into other descriptions via reference to *external* units or locally declared *units* as

$$\$ < unit\_name >:< unit\_type > (< actual\_parameter\_list >)$$

In a hierarchical description several nesting levels can be present. The user may declare his own units, as well as use the ones previously defined and stored in a Unit Data Base.

From a semantic point of view, the reference to other units with correct parameter passing is implicit in the substitutions and simplifications by demodulation prior to the verification process.

5.3 Axiomatizing circuit descriptions

Sets of description-specific axioms formally express the specified and the implemented *functionality* of the circuit. Referring to the well-defined semantics associated with each language primitive, each HDL statement is associated with a nested composition of the corresponding First-Order Logic formulæ. In particular,

- if the description is structured as a `-CASE-` statement on the *current state*, this is transformed to a set of
  `if (eq(selector,state1),` $< activities\_state\_1 >$,
  `if(eq(selector, state2),` $< activities\_state\_2 >, \ldots < activities\_state\_n >$ `)`
  `...)).`
- for each state, if *preconditions* must be satisfied by the status variables to enable the execution of the concurrent data path operation set, the description contains (nested) `-IF-` statements whose semantics is rendered as
  `if (`$< bool\_exp\_1 >, < activity\_list\_1 >$,
  `if(` $< bool\_exp\_2 >, < activity\_list\_2 >, \ldots < activity\_list\_n >$`) ...)).`
- if the body is a set of `transfer` and `connect` activities, their corresponding semantics is given as a First-Order Logic expression in the OTTER form. In the first case, the access to a past value requires the introduction of a P functional that includes the entire right side expression,
- `indexing`, `addressing`, and `concatenation` on operands is expressed via a composition of LISP-like *select* and *append* recursive semantic functions,
- body descriptions of *locally declared* `-UNITS-` are handled as additional circuit descriptions,
- calls to `-EXTERNAL-` units require to complete the semantics by including from a library of descriptions the formulæ expressin g their functionality,
- calls to (local or external) units require *parameter passing*, which amounts to correct substitutions of the arguments of OTTER's First-Order Logic expressions. This is automatically performed as a set of rewrites, with a check on type and number of parameters.

These criteria for semantic extraction are easily automatizable in an interpreter from ART$*$ to OTTER's First-Order Logic formulation, since the mapping between the ART$*$ primitives and their corresponding semantic functions is univocally determined.

6 Example: a 16-bit counter

To illustrate the proposed verification methodology and to compare the results obtained by different description and verifcation approaches, a simple example has been chosen: a 16-bit counter with enable and synchronous clear.

RT-level descriptions in ART$*$ The ART$*$ description of its behavioural specification is given in Fig. 4.

Its implementation is designed as a composition of four SN74LS163 4-bit synchronous up-counters, as schematically outlined in Fig. 5. The ART$*$ structure-oriented description is given on Fig. 6. The corresponding OTTER semantics is just a correct connection of calls to the SN74LS163 module, while the behavioural description of the 4-bit module closely resembles the one of the overall counter.

```
" Synchronous 16-bit up-counter: behavioural description"

-INPUT-         INP[15:0], CLEAR, ENABLE, LOAD, CK;
-OUTPUT-        OUT[15:0], CARRY;
-REGISTER-      F[15:0];

-BEGIN-

-IF- CK -THEN-
   -IF- CLEAR' -THEN-        "if clear = 0, then load F with 16 zeros"
        F<=#OB16
   -ELSE-                    "clear = 1, load = 0"
      -IF- LOAD' -THEN-
           F<=INP
      -ELSE-
         -IF- ENABLE -THEN-  "clear = 1, load = 1"
              F<=INC(F)      "increment F "
         -FI-
      -FI-
   -FI-
-FI-,

OUT = F,
CARRY = (*F)*ENABLE         "CARRY is the terminal count, * denotes the AND oper
   ator"

-FINIS-
```

Figure 4: ART∗ behavioural description of the 16-bit counter

The verification procedure The verification procedure is implied by the design style: since the design is hierarchical, in terms of a 4-bit counter functional block, and since the implementation strategy exploits modularity by using four cascaded instances of the same SN74LS163 chip, the verification will be both modular and hierarchical. In particular,

- the ART∗ behavioural and structural descriptions of the SN74LS163 module are validated first. Checking the *observational equivalence* of the gate-level structure vs. the functionality specified in the data book is reduced to a proof of equivalence between the First-Order formulæ that express the semantics of each description,
- the proved descriptions are then made available in a *library* of verified functional blocks, together with the corresponding behavioural and structural semantic expressions,
- the hierarchical proof at the RT level relies then on the already verified functional blocks. In this case, the description of the SN74LS163 is linked as `-EXTERNAL-` from a library of components, and the global verification is required to show that the four cascaded instances of the correct 4-bit counter exibit the behaviour specified for the 16-bit counter,
- a mixed-mode verification is possible, since either the behavioural or the structural set of formulæ may be included in the set of the 16-bit counter structural description: both possibilities have here been tried out.

The capability of handling in OTTER data path registers as a unique entity and data inputs in a symbolic way, allows to considerably reduce the number of states: data path bit slicing

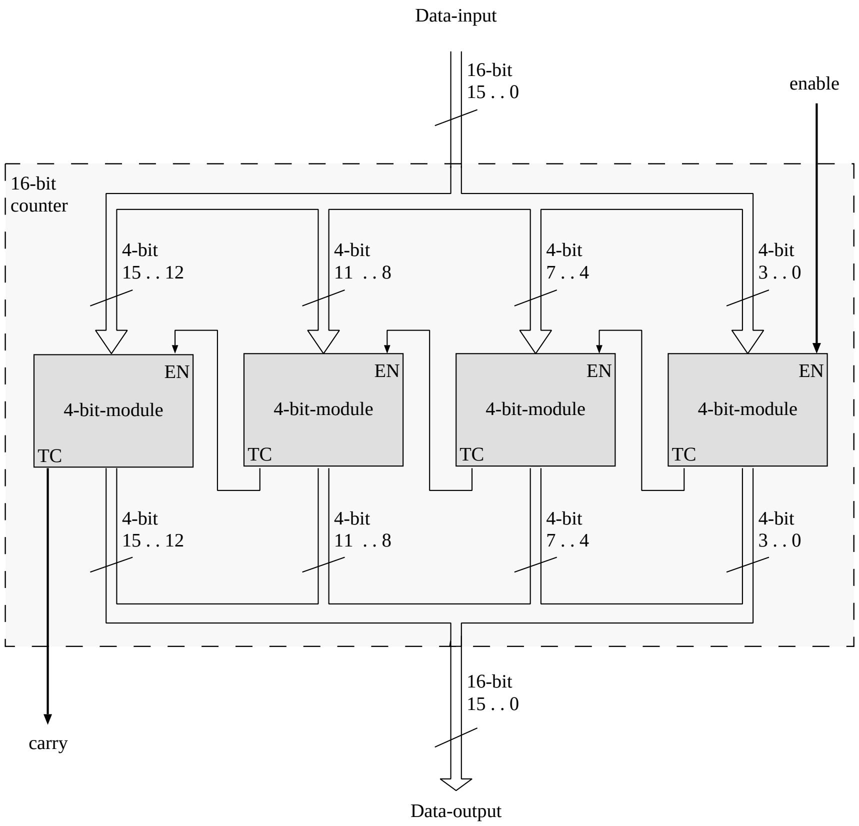

Figure 5: Modular implementation of the 16-bit counter

```
" Synchronous 16-bit up-counter: Structural description"

-INPUT-     INP[15:0], CLEAR, ENABLE, LOAD, CK;
-OUTPUT-    OUT[15:0], CARRY;
-TERMINAL-  C12, C8, C4, C0;

-EXTERNAL-  SN74LS163.FUN;  "Behavioural description"
                            "((D,C,B,A), LOAD, CLEAR, ENP, ENT, CLOCK,"
                            "(QD,QC,QB,QA), RIPCAR)"
-BEGIN-

$LSC:  SN74LS163.FUN(INP[3:0], LOAD, CLEAR, ENABLE, ENABLE, CK; OUT[3:0], C4),
$NLSC: SN74LS163.FUN(INP[7:4], LOAD, CLEAR, C4, C4, CK; OUT[7:4], C8),
$NMSC: SN74LS163.FUN(INP[11:8], LOAD, CLEAR, C8, C8, CK; OUT[11:8], C12),
$MSC:  SN74LS163.FUN(INP[15:12], LOAD, CLEAR, C12, C12, CK; OUT[15:12], C0),
CARRY = C0

-FINIS-
```

Figure 6: ART∗ structural description of the 16-bit counter

and subsequent Shannon decomposition (standard pr oof techniques in the known RT level verification approaches), is therefore completely avoided and no enumeration on the data/control inputs is necessary.

Comparing different verification strategies Starting from the reported ART∗ descriptions, a comparison has been drawn on the efficiency of four different verification strategies.

1. To show the influence of syntactic data abstraction,
   - a completely *bit-sliced* data path, with symbolic handling of inputs (columns 4-5),
   - 16-bit or 4-bit input/output *vectors* handled as a unique data structure (columns 2-3)

   are compared (Tab. 2).
2. In addition to that, the influence on the performance of compositionality in the semantics is shown for each of the above alternatives, through the influence on the performance due to the abstraction level of the submodule semantics:
   - results in columns 2 and 4 are obtained with the inclusion of the *behavioural semantics* of the SN74LS163 in the overall structural description,
   - columns 3 and 5 are relative to a *structural semantics* of the 4-bit counter.

   In the first case, a *mixed-mode* verification results.

Results All four proofs have been run with the same proof strategy (based on pure rewrites) and the same OTTER-specific optimization modalities, in a completely symbolic way, on a SUN *SparcStation II*. Results are therefore directly comparable. From the results, it follows that (Tab. 2)

1. as for syntactic data abstraction, being able to treat data bit_vectors as a single entity rather than resorting to slicing is an evident advantage: expressions involving the data

Performance parameter	vectorial		sliced	
	behav.	struct.	behav.	struct.
total CPU time (s)	6.90	24.18	20.44	62.18
run time	7.08	24.34	20.62	62.35
system time	0.18	0.14	0.11	0.32
input time	0.24	0.57	0.37	0.71
hyper_res time	0.06	0.04	0.04	0.05
demod time	6.21	23.17	19.64	61.07
print_cl time	0.14	0.14	0.15	0.11
memory used (Kbytes)	283.3	321.2	344.9	468.7
total # of clauses	97	138	99	157
clauses input	63	122	83	141
clauses generated	16	16	16	16
# of rewrites	26092	52605	59017	109071

Table 2: Comparative results

path are considerably simplified. In this case, considering a 16-bit data path as formed by four 4-bit vectors rather than 16 slices has brought to reducing the time devoted to rewrites (demod. time) to about one third (columns 2 and 3 vs. columns 4 and 5) while the number of rewrites (demod. steps) is more than halved,

2. as for compositionality in the semantics, the possibility of mixing behavioural portions in a basically structural semantics leads to a significant performance improvement: the reasoning (demod.) time needed in the mixed-mode approach (columns 2 and 4 vs. columns 3 and 5) is brought to a mere 26.8% and 32%, resp ectively, and the number of rewrites is halved.

Considering both effects together, memory needs do not vary as significantly as the time performance: the memory allocation is reduced to the 60% in the best case, while the overall CPU time required for achieving the proof is lowered to a mere 11.4% (col. 5 vs. col. 2). Since the pre- and postprocessing times are almost constant (between 0.5 s and 1 s), the effective reasoning (hyperresolution plus demodulation time) amounts to 88.6% (col. 2) and 98% (col. 5) of the overall CPU time.

Due to the structural composition of the four modules in a cascaded chain, in this example it has been necessary to include the complete set of formulæ relative to the 4-bit counter. In general, if only some portions of a module were required to describe the functionality of the RT-level system, only the needed First-Order expressions would appear in the RT-level description. This would result in a reduction in the number of clauses and memory occupation, but automating the individuation of the minimum nucleus of formulæ/ is not a straightforward task.

7 Conclusions

The paper has presented a framework for the efficient verification of complex FSM's in which RT level behavioural and structural descriptions of a circuit are compared at a common semantic level. The strength of the proposed verification method lies in the definition of a compositional semantics that encompasses both gate and RT abstraction levels. The compositionality is ensured by construction: it is directly implied by the incremental definition of the RT level semantics as an extension of the gate level semantics, and the formalization of the ART∗ primitives in terms of RT level basic functions.

The framework, which is based on semantic definitions in terms of First-Order theories, has been demonstrated by axiomatizing the hardware description language ART∗, and subsequently applying the proposed verification methodology: proving the correctness of a circuit with respect to its specifications is done by establishing logical equivalence between their semantics, given in terms of First-Order formulæ. This methodology allows to efficiently exploit the hierarchy and structural regularity in a design to conduct a hierarchical and modular proof, while the common semantic domain enables mixed-mode verification.

The verification methodology has been implemented in OTTER, a general-purpose theorem prover for First-Order Logic with equality, based on resolution techniques. The efficiency of the approach has been demonstrated through a comparison with the implementations in OTTER of traditional RT-level verification techniques, showing a run-time improvement of a factor 10 in the reasonment time and a factor 4 in the reasonment steps, which is due to the exploitation of both syntactic data abstraction and compositionality in the semantics.

Acknowledgments

The author would like to thank Prof. Paolo Prinetto and Prof. Dr. Bernhard Steffen for many clarifying discussions and the help in reviewing the present work.

References

[Back78] J. Backus:
"Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs,"
Communications of the ACM, Vol. 21, N. 8, August 1978, pp. 615-641.

[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.

[BRBr90] K. Brace, R. Rudell, R. Bryant:
"Efficient implementation of a BDD package,"
DAC-27 : 27th ACM/IEEE Desing Automation Conference, Orlando, FL (USA), June 1990, pp. 40-45.

[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.

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

[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.

[Glus68] V. Glushkov:
"Automation theory and formal microprogram transformation,"
Cybernetics, pp.1-8, January 1968.

[Knob81] H.J. Knobloch:
"Description and simulation of complex digital systems by means of the Register Transfer level language RTS Ia,"
Computer Design aids for VLSI Circuits, NATO Advanced Study Series E N. 48, Sijthoff & Noordhoff, 1981.

[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.

[McCl86] E.J. McCluskey:
"Logic design principle with emphasis on testable semicustom circuits,"
Prentice-Hall, Englewood Cliffs, NJ (USA), 1986.

[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).

[NKFT90] H. Nakamura, Y. Kukimoto, M. Fujita, H. Tanaka:
"A data path verifier for Register Transfer level using Temporal Logic language TOKIO,"
CAV'90 - Workshop on "Computer-Aided Verification", DIMACS Series in Discrete Mathematics and Th. Computer Science, Vol.3, AMS, 1990, pp. 493-504.

[Pail86] J.L. Paillet:
"A functional model for descriptions and specifications of digital devices,"
IFIP WG 10.2 Workshop "From HDL descriptions to guaranteed correct circuit designs", Grenoble (France), September 1986.

[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.

[Vemu89] R. Vemuri:
"A formal model for Regiater-Transfer level structures and its applications in verification and synthesis,"
IMEC-IFIP Workshop on "Applied Formal Methods For Correct VLSI Design", Leuven (Belgium), November 1989, Vol. 2, pp. 77-96.

8 Appendix

Here are listed the basic semantic functions corresponding to the ART∗ primitives. The adopted syntax is the First-Order Logic accepted as input by the OTTER theorem prover.
As a convention, identifiers starting with letters `x, y, w, z` denote variables, all others represent constants.

```
% Definition of a vector for bus connections:

(bus([xbuswidth, [xsourcebits, xsource_name, xenable]]| x_other_connections]])).

% selection of a single element in a bitvector:

(sel([x|xr], xi) = $IF($EQ(xi,0),x,el(xr,$SUM(xi,-1)))).
(sel([],x) = Element_out_of_range).

% extraction of a subrange of a bitvector:

(sel_vec(xvec, first, 0) = []).
(sel_vec(xvec, first, range) =
     [el(xvec(first)|sel_vec(xvec,$SUM(first,1), $DIFF(range,1)]).

% extraction of a submatrix:

(sel_mat(xmat, first_r, range_r, first_c, range_c) =
     sel_submat(sel_vec(xmat,first_r, range_r), first_c, range_c)).

(sel_submat([x|xr],first_c, range_c) =
     [sel_vec(x,first_c,range_c|sel_submat([xr],first_c, range_c)]).

(sel_submat([], first_c, range_c) = []).

% concatenation functions:

(append([], x) = x).
(append([x|xr], y) = [x|append(xr,y)]).
(append_vm([x1|xr], [y1|yr]) = [append(x1,y1)|append_vm(xr,yr)]).

% transpose to column vector:

(to_col(vec) = [vec]).

% complementing a matrix:

(compl_mat([xmat1|xmatr]) = [compl_vec(xmat1)|compl_mat(xmatr)]).
(compl_mat([]) = []).

% complementing a vector:

(compl_vec([xvec1|xvecr]) = [not(xvec1)|compl_vec(xvecr)]).
```



```
(compl_vec([]) = []).

% complementing predefined values:

(not(1) = 0).
(not(0) = 1).
(not(d) = d).
(not(Vcc) = Gnd).
(not(Gnd) = Vcc).
(not(Float) = Float).
(not(Nc) = Nc).

% unary boolean operators:

(cross_and([0]) = 0).
(cross_and([1]) = 1).
(cross_and([0|xr]) = 0).
(cross_and([1|xr]) = cross_and(xr)).

(cross_or([0]) = 0).
(cross_or([1]) = 1).
(cross_or([1|xr]) = 1).
(cross_or([0|xr]) = cross_or(xr)).

(cross_xor([x]) = x).
(cross_xor([1|xr]) = not(cross_xor(xr))).
(cross_xor([0|xr]) = cross_xor(xr)).

% binary boolean operator AND:

(and_mat([xrow1|xmr], [yrow1|ymr]) = [and_vec([xrow1, yrow1])|and_mat(xmr,ymr)])
   .
(and_mat([],[]) = []).

(and_vec([x1|xvr], [y1|yvr]) = [and(x1, y1)|and_vec(xvr,yvr)]).
(and_vec([],[]) = []).

% multinary boolean function NAND:

(nand_mat([xmat1|xmatr]) = compl_mat(and_mat(xmat1, _and_mat(xmatr))).
(_and_mat([xmat1|xmatr]) = and_mat(xmat1, _and_mat(xmatr))).
(_and_mat([xmat1] = xmat1).

(nand_vec([xvec1|xvecr]) = compl_vec(and_vec(xvec1, _and_vec(xvecr))).
(_and_vec([xvec1|xvecr]) = and_vec(xvec1, _and_vec(xvecr))).
(_and_vec([xvec1] = xvec1).

(nand_s([x1|xr]) = not(and(x1, _and(xr))).
(_and([x1|xr]) = and(x1, _and(xr))).
(_and([x1,x2]) = and(x1,x2)).
```

```
% Arithmetic functions:

(adc_msbr(x, y, xcarry) = _add_msbr(x, y, [], xcarry)).

(add_msbr(x, y) = _add_msbr(x, y, [], 0)).

(_add_msbr([x|xr], [y|yr], xsum, xcarry) =
        = _add_msbr(xr, yr, append(xsum, [bit_add(x, y, xcarry)]),
                    bit_carry(x, y, xcarry))).

(_add_msbr([], [], xsum, xcarry) = append(xsum, [xcarry])).
(_add_msbr([], [y|yr], xsum, xcarry) = _add_msbr([xcarry], [y|yr], xsum, 0)).
(_add_msbr([x|xr], [], xsum, xcarry) = _add_msbr([], [x|xr], xsum, xcarry)).

(add_msbl(x, y) = reverse(add_msbr(reverse(x), reverse(y)))).

(reverse([]) = []).
(reverse([x|y]) = append(reverse(y), [x])).

% Increment and decrement functions INC, DEC:

(inc(xv1) = resize(adc(xv1, fan(0, length(xv1), 1)), length(xv1))).
(dcr(xv1) = resize(sbb(xv1, fan(0, length(xv1), 1)), length(xv1))).

% F: fan-out function

(fan(xval, xlength) = [xval|fan(xval, $diff(xlength,1))]).
(fan(xval, 0) = []).

% Shift functions:

% ROR : Rotate right

(ror_1(xvec) = [sel(xvec, $DIFF(length(xvec),1))|
                sel_vec(xvec, 0, $DIFF(length(xvec),1))]).
(ror(xvec,xn) = append(sel_vec(xvec, $DIFF(length(xvec),xk), xk),
                      sel_vec(xvec, 0, $DIFF(length(xvec,xk))))).

% ROL : Rotate left

(rol_1([x1|xvecr]) = append(xr, [x])).
(rol(xvec,xn) = append(sel_vec(xvec, xk, $DIFF(length(xvec),xk)),
                      sel_vec(xvec, 0, xk))).
% RSH: Right shift

(rsh_1(xs,xvec) = [xs|sel_vec(xvec, 0, $DIFF(length(xvec),1)))]).
(rsh(xs,xvec,xk) = append(fan(xs,xk),
                         sel_vec(xvec, 0, $DIFF(length(xvec,xk))))).

% RSH: Right shift
```



```
(lsh_1(xs,xvec) = append(xr,[xs])).
(lsh(xs,xvec,xn) = append(sel_vec(xvec, xk, $DIFF(length(xvec, xk)),
                          fan(xs, xk))).

% Example of arithmetic theorems:

(inc(dcr(x)) = x).

(rol_1(ror_1(xvec) = xvec).
(rsh_1(xs,lsh_1(xs,xvec), xvec) = xvec).

% partial cancellation of rotate:
% if (xn < xm), then

(and(rol(ror(xvec,xn), xm), $GE(xm,xn)) = rol(xvec,$DIFF(xm,xn))).

% Compare functions:

% For all structures

(eq(x,y) = $ID(x,y)).
(neq(x,y) = not(eq(x,y))).

% On bitvectors:

(gt(xvec1,xvec2) = el(sbb(xvec2,xvec1,0),0)).
(lt(xvec1,xvec2) = el(sbb(xvec1,xvec2,0),0)).

(ge(xvec1,xvec2) = not(lt(xvec1,xvec2))).
(le(xvec1,xvec2) = not(gt(xvec1,xvec2))).
```

Aachener Informatik-Berichte

Please write to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, W-5100 Aachen

87-1	Jahresbericht 1986
87-2	D. de Frutos Escrig / K. Indermark: Equivalence relations of non-deterministic Ianov-schemes
87-3	M. Nagl: A Software Development Environment Based on Graph Technology
87-4	C. Lewerentz / M. Nagl / B. Westfechtel: On Integration Mechanisms within a Graph-based Software Development Environment
87-5	R. Rinn: Über Eingabeanomalien bei verschiedenen Inferenzmodellen
87-6	W. Damm / G. Döhmen: Specifying Distributed Computer Architectures in AADL*
87-7	G. Engels / C. Lewerentz / W. Schäfer: Graph Grammar Engineering: A Software Specification Method
87-8	M. Nagl: Set Theoretic Approaches to Graph Grammars
87-9	C. Lewerentz / A. Schürr: Experiences with a Database System for Software Documents
87-10	H. Klaeren / K. Indermark: A New Implementation Technique for Recursive Function Definitions
87-11	R. Loogen: Design of a parallel programmable graph reduction machine with distributed memory
87-12	J. Börstler / U. Möncke / R. Wilhelm: Table Compression for Tree Automata
87-13	C. Lewerentz: Extended Programming in the Large in a Software Development Environment
87-14	G. Engels / Th. Janning / W. Schäfer: A Highly Integrated Tool Set for Program Development Support
87-15	R. Loogen / U. Goltz: A non-interleaving semantic model for nondeterministic concurrent processes
87-16	Th. Hafer: On the Boolean Closure of Büchi Tree Automaton Definable Sets of w-Trees.
87-17	U. Heuter: First-Order Properties of Trees, Star-Free Expressions and Aperiodicity
88-1	G. Esser / J. Rückert / F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik
88-2	P. Martini / O. Spaniol: Token-Passing in High-Speed Backbone Networks for Campus-Wide Environments
88-3	Th. Welzel: Simulation of a Multiple Token Ring Backbone
88-4	P. Martini: Performance Comparison for HSLAN Media Access Protocols
88-5	P. Martini: Performance Analysis of Multiple Token Rings
88-6	A. Mann / J. Rückert / O. Spaniol: Datenfunknetze
88-7	A. Mann / J. Rückert: Packet Radio Networks for Data Exchange

88-8	A. Mann / J. Rückert: Concurrent Slot Assignment Protocol for Packet Radio Networks
88-9	W. Kremer / F. Reichert / J. Rückert / A. Mann: Entwurf einer Netzwerktopologie für ein Mobilfunknetz zur Unterstützung des öffentlichen Straßenverkehrs
88-10	K. Jakobs: Towards User-Friendly Networking
88-11	K. Jakobs: The Directory - Evolution of a Standard
88-12	K. Jakobs: Directory Services in Distributed Systems - A Survey
88-13	M. Schümmer: RS-511, a Protocol for the Plant Floor
88-14	U. Quernheim: Satellite Communication Protocols - A Performance Comparison Considering On-Board Processing
88-15	P. Martini / O. Spaniol / Th. Welzel: File Transfer in High Speed Token Ring Networks: Performance Evaluation by Approximate Analysis and Simulation
88-16	Jahresbericht 1987
88-17	W. Thomas: Automata on Infinite Objects
88-18	M. Sonnenschein: On Petri Nets and Data Flow Graphs
88-19	H. Vogler: Functional Description of the Contextual Analysis in Block-Structured Programming Languages: A Case Study of Tree Transducers
88-20	Th. Welzel: Einsatz des Simulationswerkzeuges QNAP2 zur Leistungsbewertung von Kommunikationsprotokollen
88-21	Th. Janning / C. Lewerentz: Integrated Project Team Management in a Software Development Environment
88-22	J. Engelfriet / H. Vogler: Modular Tree Transducers
88-23	W. Thomas: Automata and Quantifier Hierarchies
88-24	U. Heuter: Generalized Definite Tree Languages
89-1	Jahresbericht 1988
89-2	G. Esser / J. Rückert / F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik
89-3	H. Vogler: Bottom-Up Computation of Primitive Recursive Tree Functions
89-4	A. Schürr: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language
89-5	J. Börstler: Wiederverwendbarkeit und Softwareentwicklung - Probleme, Lösungsansätze und Bibliographie
89-6	K. Jakobs: OSI - An Appropriate Basis for Group Communication?
89-7	K. Jakobs: ISO's Directory Proposal Evolution, Current Status and Future Problems
89-8	B. Westfechtel: Extension of a Graph Storage for Software Documents with Primitives for Undo / Redo and Revision Control
89-9	P. Martini: High Speed Local Area Networks - A Tutorial
89-00	Jozef Filser: Bolidische Gedangen (Gabidel 5): "Ieber die Kombjuderkunzt"
89-10	P. Davids / Th. Welzel: Performance Analysis of DQDB Based on Simulation
89-11	M. Nagl (Ed.): Abstracts of Talks presented at the WG '89 15th International Workshop on Graphtheoretic Concepts in Computer Science, Rolduc (NL) near Aachen (D), June 14-16, 1989.

89-12 P. Martini: The DQDB Protocol - Is it Playing the Game?

89-13 M. Schümmer: CNC / DNC Communication with MAP

89-14 M. Schümmer: Local Area Networks for Manufactoring Environments with hard Real-Time Requirements

89-15 M. Schümmer / Th. Welzel / P. Martini: Integration of Field Bus and MAP Networks - Hierarchical Communication Systems in Production Environments

89-16 G. Vossen / K.-U. Witt: SUXESS: Towards a Sound Unification of Extensions of the Relational Data Model

89-17 J. Derissen / P. Hruschka / M.v.d.Beeck / Th. Janning / M. Nagl: Integrating Structured Analysis and Information Modelling

89-18 A. Maaßen: Programming with Higher Order Functions

89-19 M. Rodriguez-Artalejo / H. Vogler: Note on A Narrowing Machine for Syntax Directed BABEL

89-20 H. Kuchen / R. Loogen / J. J. Moreno-Navarro / M. Rodriguez-Artalejo Graph-based Implementation of a Functional Logic Language

89-i4 O. Spaniol: Natürliche Intelligenz oder: Der Parallel-Igel

89-21 G. Vossen / S. Wang: Correct and Efficient Deadlock Detection and Resolution in Distributed Database Systems

89-22 G. Vossen / R. Schwarz: Dynamic Scheduling of Transactions in Design Databases based on Version Consistency

90-1 Jahresbericht 1989

90-2 V. Jansen / A. Potthoff / W. Thomas / U. Wermuth: A Short Guide to the AMORE System (computing Automata, MOnoids, and Regular Expressions)

90-3 J. Skurczynski: On Three Hierarchies of Weak SkS Formulas

90-4 R. Loogen: Stack-based Implementation of Narrowing

90-5 H. Kuchen / A. Wagener: Comparison of Dynamic Load Balancing Strategies

90-6 K. Jakobs / F. Reichert: Directory Services for Mobile Communication

90-7 K. Jakobs: What's Beyond the Interface-OSI Networks to Support CooperativeWork

90-8 K. Jakobs: Directory Names and Schema - An Evaluation

90-9 U. Quernheim / D. Kreuer: Das CCITT - Signalisierungssystem Nr. 7 auf Satellitenstrecken; Simulation der Zeichengabestrecke

90-10 U. Quernheim / St. Jacobs / D. Kreuer: Adaptive ARQ-Fehlersicherungsverfahren für die Point-to-Multipoint-Satellitenkommunikation

90-11 J.J.Moreno-Navarro / H. Kuchen / R. Loogen / M. Rodriguez-Artalejo: Lazy Narrowing in a Graph Machine

90-12 K. Jakobs / J. Kaltwasser / F. Reichert / O. Spaniol: Der Computer fährt mit

90-13 R. Mathar / A. Mann: Analyzing a Distributed Slot Assignment Protocol by Markov Chains

90-14 A. Maaßen: Compilerentwicklung in Miranda - ein Praktikum in funktionaler Programmierung

90-15 M. Nagl / A. Schürr: A Specification Environment for Graph Grammars

90-16 A. Schürr: PROGRESS: a VHL-Language Based on Graph Grammars

90-17 M. Möller: Ein Ebenenmodell wissensbasierter Konsultationen - Unterstützung für Wissensakquisition und Erklärungsfähigkeit

90-18 E. Kowalewski: Entwurf und Interpretation einer Sprache zur Beschreibung von Konsultationsphasen in Expertensystemen

90-19 F. Strauß: Erklärungsfähigkeit eines ebenenorientierten Expertensystemmodells

90-20 Y. Ortega-Mallen / D. de Frutos-Escrig: A Complete Proof System for Timed Observations

90-21 M. Nagl: Modelling of Software Architectures: Importance, Notions, Experiences

90-22 H. Faßbender / H. Vogler: A Call-by-need Implementation of Syntax Directed Functional Programming

90-23 Th. Janning / M. Lefering: A Transformation from Requirements Engineering into Design - The Method and the Tool

90-24 J. Noack: Natürlichsprachliche Interaktion mit Datenbanken

90-25 R. Mathar: Concurrence Probabilities for a Locally Slotted Packet Radio Network by Combinatorial Methods

90-26 F. Herrmann / Heiko Vogler: Primitive Recursive Programs for Defining Tree Functions

90-27 Z. Fülöp / F. Herrmann / S. Vagvölgyi / H. Vogler: Tree Transducers with External Functions

90-28 P. Bachmann / W. Oberschelp / B. Thalheim / G. Vossen: The Design of RAD: Towards an Interactive Toolbox for Database Design

91-1 Jahresbericht 1990

91-2 F. Beuting / F. Piron / K. Rausch: Ein neuronales Modell zur Feststellung von für die visuelle Aufmerksamkeit relevanten Figureneigenschaften

91-3 B. Steffen / A. Ingolfsdottir: Characteristic Formulae for Processes with Divergence

91-4 M. Portz: A new class of cryptosystems based on interconnection networks

91-5 H. Kuchen / G. Geiler: Distributed Applicative Arrays

91-6 L. Staiger: Kolmogorov Complexity and Hausdorff Dimension

91-7 L. Staiger: Syntactic congruences for ω-languages

91-8 K. Jakobs / K. D. Kleefeld: Using Public Communication Services for Archaeological Applications

91-9 E. Kuikka: A Proposal for a Syntax-Directed Text Processing System

91-10 K. Gladitz / H. Faßbender / H. Vogler: Compiler-based Implementation of Syntax-Directed Functional Programming

91-11 R. Loogen / S. Winkler: Dynamic Detection of Determinism in Functional Logic Languages

91-12 K. Indermark / M. Rodríguez Artalejo (Eds.): Granada Workshop on the Integration of Functional and Logic Programming

91-13 R. Hager / W. Kremer: The Adaptive Priority Scheduler: A More Fair Priority Service Discipline.

91-14 A. Fasbender / W. Kremer: A New Approximation Algorithm for Tandem Networks with Priority Nodes.

91-15 J. Börstler / A. Zündorf: Revisiting extensions to Modula-2 to improve reusability.

91-16 J. Börstler / T. Janning: Bridging the gap between Requirements Analysis and Design

91-17 A. Zündorf / A. Schürr: Nondeterministic Control Structures for Graph Rewriting Systems

91-18 M. Jarke / J. Mylopoulos / J. W. Schmidt / Y. Vassiliou: DAIDA: An Environment for Evolving Information Systems

91-19 M. Jeusfeld / M. Jarke: From Relational to Object-Oriented Integrity Simplification

91-20 G. Hogen / A. Kindler / R. Loogen: Automatic Parallelization of Lazy Functional Programs

91-21 O. Spaniol: ODP (Open Distributed Processing): Yet another Viewpoint

91-22 H. Kuchen / F. Lücking / H. Stoltze: The Topology Description Language TDL

91-23 S. Graf / B. Steffen: Compositional Minimization of Finite State Systems

91-24 R. Cleaveland / J. Parrow / B. Steffen: The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems

91-25 R. Mathar/J. Mattfeldt: Optimal Transmission Ranges for Mobile Communication in Linear Multihop Packet Radio Networks

91-26 M. Jeusfeld/ M. Staudt: Query Optimization in Deduktive Object Bases

91-27 J. Knoop/B. Steffen: The Interprocedural Coincidence Theorem

91-28 J. Knoop/B. Steffen: Unifying Strength Reduction and Semantic Code Motion

91-29 R. Cleaveland/M. Dreim"uller/B. Steffen: Faster Model Checking in the Model Mu-Calculus

91-30 T. Margaria: First-Order Theories for the Verification of Complex FSMs