# Data Type Analysis for Hardware Synthesis from Object-Oriented Models

Martin Radetzki, Ansgar Stammermann, Wolfram Putzke-Röming, Wolfgang Nebel
OFFIS Research Center, Escherweg 2, 26121 Oldenburg, Germany
e-mail: radetzki@offis.uni-oldenburg.de

## Abstract

*Object-oriented modeling of hardware promises to help deal with design complexity through higher abstraction and better support for reuse. Whereas simulation of such models is rather easy to achieve, synthesis turns out to require the application of quite sophisticated techniques. In this paper, we devise a solution of the foremost problem, optimized synthesis of object-oriented data types. The outlined algorithms have been implemented for an object-oriented dialect of VHDL and may also contribute, possibly in a co-design context, to synthesis from languages such as C++ or Java. We explain our synthesis methods and show their impact with the example of a microprocessor model.*

## 1 Introduction

The need to keep productivity up with the growing complexity of hardware units under design has been and continues to be among the most pressing design methodology issues. It has been effectively addressed by design at ever-higher abstraction, from gate via register-transfer to behavioral level, supported by the application of synthesis tools to hardware description languages such as VHDL [9]. In this progress, the use of object-oriented techniques, proven in software engineering [5], in hardware design could be the next step not only to increase the abstraction, but also the potential for reuse of hardware models [1]. Moreover, it could also contribute to a unified representation supporting the co-design of hardware and software [10].

Since the early 90s, a variety of proposals have been made to extend VHDL for object-orientation, among them [3, 11, 12] concentrating on object-oriented data types and [13] on VHDL design entities. Also a language integrating some object-oriented concepts with a Verilog-like syntax has been reported on [6]. Other authors used object-oriented software programming languages such as C++ to model hardware ([10], chapter 9). The focus, so far, has largely been on modeling for simulation of system functionality and performance evaluation.

Taking such initial model as a specification, it is an obvious approach to manually refine it to the degree of detail required for hardware synthesis. Currently, this means designers would have to remove all object-oriented constructs, which is not only a tedious and error-prone task, but also requires a costly transition to a totally different conceptual model. Therefore, capability of automatic synthesis directly from object-oriented models is highly desirable.

Another motivation to deal with synthesis of object-oriented constructs is the increasing use of software languages, e.g. dialects of C such as the recently commercialized Handel-C [4], to allow a more software-like specification of hardware in the fields of reconfigurable computing and hardware-software-systems. An extension of this approach to utilize the benefits of, say, C++ or Java would be a natural step. For instance, use of Java for hardware-software-specification has recently been reported on at past ICCAD [7]. Also, there is research on language-independent object-oriented co-specification [8].

Results of our research on optimized synthesis from object-oriented hardware models will be presented in the remainder of this paper. The next section is devoted to a detailed description of the major problems to be addressed and will introduce an example to be used for further motivations and explanations. Analysis methods explained in the third section will be the basis for optimized resource allocation outlined in section four. The implementation of algorithms for an object-oriented dialect of VHDL, *Objective VHDL* [11], and results obtained therefrom are described in the fifth and sixth section, respectively, and lead us to some conclusions regarding the state of our research and necessity of further work.

## 2 Problem formulation

Basic to object-orientation is the concept of classes comprising attributes (data fields) and operations (methods). An object is an instance of a class. Classes can be derived from each other. A derived class inherits attributes and operations from its parent, which allows to factor out common properties of the derived classes for a single implementation in the parent. Moreover, the derived class can have additional properties or redeclare an inherited operation.

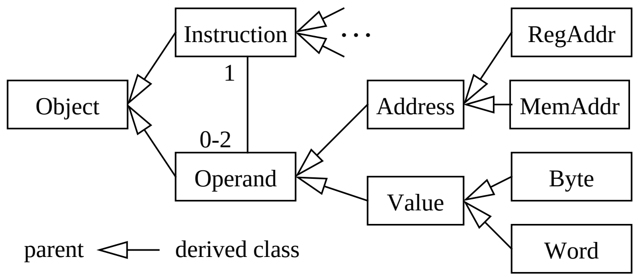


**Fig. 1: Inheritance hierarchy of μP data**

Consider, for example, data types occurring in a microprocessor (Fig. 1). Basically, we have instructions and operands. An instruction may have zero, one, or two operands as attributes. An operand may be an address or a value. An address may address a register (RegAddr) or a memory location (MemAddr). Also, there are different types of values, e.g. Byte and Word.

Polymorphism comes into play when related classes have to be handled in a uniform way. For instance, a data field 'operand' of an instruction may either be an address or a direct value. Hence, it must be able to hold not only class Operand, but also all classes derived from Operand. Likewise, we want to store both, instructions and operands, in a memory. To this end, we derive both from a parent class, Object, and declare the memory so as to hold instances of Object and all derived classes (cf. Fig. 2). An instruction register is another example; it should be able to contain any concrete instruction such as LOAD or ADD derived from class Instruction.

In software, polymorphism is easily implemented with references which all have the same size. However, considering the instruction register in hardware, we certainly do not want it to contain a reference to some instruction stored elsewhere. Instead, the register should itself be able to contain any physical instruction. As well, a register in the register file should contain an operand, and an address register should contain an address, not a respective reference. This, however, imposes the task of determining the storage space required for any of these polymorphic objects when synthesizing hardware from an object-oriented model.

Conceptually, a memory declared polymorphic with class type Object must be able to also store any class derived from Object. But some of these classes might not occur in the system because no objects are created from them (e.g. Object, which is only used to declare a common memory for instructions and operands) or only be used in another system which (re-)uses the same inheritance hierarchy. And some classes might, according to data flow (see Fig. 2), occur only in some part of the system. For instance, in the memory, and in the register file as well, there will never be a register address since this class only occurs as part of an instruction. Or, in a RISC processor, the operand within an instruction may only be of class RegAddr, not MemAddr. The aim of this work is to utilize the resulting storage space optimization potential.

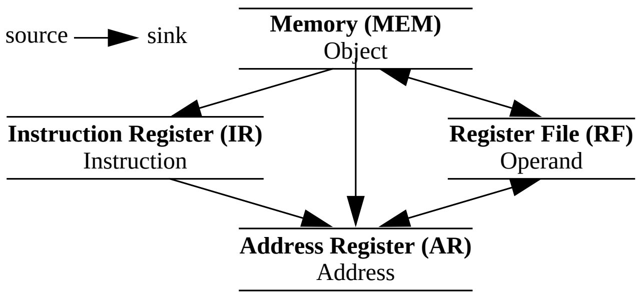


**Fig. 2: Data flow in μP model**

We therefore define static data type analysis as the task of determining the class types that may have to be held in a polymorphic object through inspection of data flow and program structure of an object-oriented hardware model.

## 3 Data type analysis

In case of objects, e.g. VHDL variables, accessed only from a single sequential part of a hardware model, data type analysis can be performed using classic data flow analysis techniques. Concurrent access to an object, as possible with VHDL signals, does however require the application of other, more pessimistic, algorithms due to statically unpredictable interleaving of concurrent reads and writes. Both aspects, as well as the combination of the developed analysis methods, will be addressed in the following. Basic to the presentation are the following terms:

*Definition 1:*

Be $Obj = \{obj_i \mid i = 1, ..., n\}$ the set of polymorphic objects of a hardware model under consideration. Be *Type* the set of all class types occurring in the inheritance hierarchies used.

- The *type vector* $T \subseteq Type^n$ is the vector whose *i*-th element $T_i$ is the set of types to be contained by $obj_i$.
- The *constraint vector* $C \subseteq Type^{..}$ is the vector whose *i*-th element is the set of types to be maximally contained by $obj_i$, i.e. the class of $obj_i$ and all its derived classes.

*Example 1:* Sets and vectors corresponding to Fig. 1 and Fig. 2

- *Obj* = {MEM, IR, AR, RF}
- *Type* = { Object, Instruction, Operand, Address, RegAddr, MemAddr, Value, Byte, Word }
- $C_{RF}$ = {Operand, Address, RegAddr, MemAddr, Value, Byte, Word} ($C_{RF}$: element of *C* corresponding to RF)
- $T_{RF}$ = { MemAddr, Byte, Word } as analysis result (see sect. 6)

### 3.1 Sequential objects

If *Obj* are sequential objects, the adaptation of data flow analysis (DFA) techniques [2] to our problem allows to determine the sets of types $T_i^{out}$, $i = 1, ..., n$, which are potentially contained by objects $obj_i$ after execution of a statement *S* when the analogue sets $T_i^{in}$ before the statement are known. This leads to the following syntax-directed formulation of data flow equations for the four basic sequential statements shown in Fig. 3:

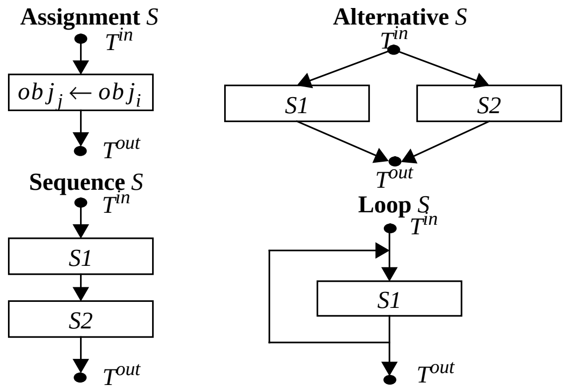


**Fig. 3: DFA of sequential statements**

- After an assignment S from $obj_i$ to $obj_j$, any type contained by the source that meets the type constraint of the target may be contained by the target. Non-target objects are not affected (Eq. 1). Note: if the right hand of an assignment is an expression, it will be handled like a function call, cf. 5.3.

$$(1) \quad T^{out} = S(T^{in}) \text{ where } T_k^{out} = \begin{cases} T_k^{in} & if \quad k \neq j \\ T_i^{in} \cap C_j & if \quad k = j \end{cases}$$

- Any branch, *S1 or S2*, of an alternative (if, case) statement *S* may be executed; we don't know statically which. Thus, the effects of both branches are unified (Eq. 2). Note: if there are more than two branches, *S2* is chosen another alternative.

$$(2) \quad T^{out} = S(T^{in}) = S1(T^{in}) \cup S2(T^{in})$$

- We deal with a sequence *S* of statements *S1* followed by *S2* by using the output of the first as the input of the second statement, see Eq. 3. For a longer sequence we choose *S2* another sequence.

$$(3) \quad T^{out} = S(T^{in}) = S2(S1(T^{in}))$$

- Finally, a loop *S* as shown in Fig. 3 shall be considered. The type vector $S1(\varnothing)$ generated by the statements inside the loop may be propagated back to its start when the loop is iterated. $S1(\varnothing)$ is therefore used in addition to $T^{in}$ as input to the inner statement $S1$. For a while loop whose $S1$ need not be executed at all, we would cover the case of bypassing the loop by adding an unmodified $T^{in}$:

(4) $$T^{out} = S(T^{in}) = S1(T^{in} \cup S1(\varnothing)) \left[ \cup T^{in}, \text{while loop} \right]$$

We now demonstrate sequential DFA with a simple example (Ex.2 corresponding to Fig. 1, 2). Consider the instruction register IR, whose first operand be a RegAddr loaded into the address register AR (line 1). After that the set of types for AR is { RegAddr }. This is used to load the value of the addressed register from the register file RF into AR (line 2), after which AR may contain types { MemAddr }. Finally, this value is assigned to an address bus in line 3 so as to address memory MEM. Globally, AR must of course be able to contain both types. The point, however, is the knowledge that only the RegAddr type is used to address RF, and only MemAddr is put on the address bus, but not the respective other type.

*Example 2:* Implementation of register-indirect addressing

*(1) AR ← IR.getOp(1); – – $T_{AR}$ = { RegAddr }*
*(2) AR ← RF[ AR ]; – – $T_{AR}$ = { MemAddr }*
*(3) AddrBus ← AR;*

## 3.2 Concurrent objects

In case of objects *Obj* available to concurrent parts of a hardware description, e.g. VHDL signals which can be read and written from several processes, we cannot make statements about the sequence of accesses in general. Thus, DFA is not applicable, and we have to make the more pessimistic worst-case assumption that any type a value of is ever assigned to such concurrent object may be passed on when the object is read from. For instance, if the statements from Ex. 2 were concurrent, AR could contain a value of type RegAddr or MemAddr when assigned to AddrBus.

We handle assignments $obj_j \leftarrow obj_i$ between concurrent objects by adding the set of types contained by the source, $obj_i$, to the set of types that the target, $obj_j$, must be able to contain:

(5) $$T_j = T_j \cup (T_i \cap C_j)$$

Comparing Eq. 5 to the case $k = i$ of Eq. 1, $T_i$ is unified with, not replaced by $T_i \cap C_j$. Also, we do not have the notion of type sets $T^{in}$ before and $T^{out}$ after a statement because, as mentioned before, we do not consider a precedence relationship between concurrent assignments in static analysis. Therefore, Eq. 5 is obviously recursive. Additional indirect recursion may be introduced through other assignments; e.g. $obj_i \leftarrow obj_j$ would cause a mutual dependency between $T_i$ and $T_j$. We will apply fixed-point iteration (FPI) to yet obtain a solution.

To formalize FPI, we define the assignment matrix $A$ with elements $A_{ij} \subseteq Type$ as follows:

(6) $$A_{ij} = \begin{cases} C_j & if \;\; Obj_j \leftarrow Obj_i \;\; or \;\; i = j \\ \varnothing & else \end{cases}$$

This allows us to write Eq. 5 for all $i$ and $j$ as a single matrix multiplication with addition $\cup$ and multiplication $\cap$ :

(7) $T(t+1) = T(t) \cdot A$ where $t$ is an index for iteration.

We start iteration with types $T_i(0)$ defined by the known types of all non-polymorphic objects assigned to the polymorphic $obj_i$. Each step propagates types according to direct assignments. It takes at most $n$ steps to propagate types transitively between the $n$ objects. Thus, iteration will reach a steady state $T(t+1) = T(t)$ after no more than $t = n$ iterations.

Since matrix multiplication complexity is $O(n^2)$, FPI will take $O(n^3)$ time in worst case. It is, however, a realistic assumption that each object is the target of only a few assignments. Then a lot of entries in $A$ will be $\varnothing$, which can be considered by optimized data structures (see 5.1) and algorithms (see 5.4) to keep the problem tractable for large $n$. Moreover, we will use design hierarchy to decompose the problem into smaller ones (see 5.3).

## 3.3 Linking the algorithms

Having considered sequential and concurrent objects independently of each other up to here, we will now devise how to deal with assignments from a sequential object, $obj_i^s$, to a concurrent object, $obj_j^c$, and vice versa. The problem is that in these cases DFA depends on the type sets generated by FPI and vice versa, which would require a costly iteration of these algorithms. Key to the solution are references between sets of types allowed by a modified notion of type vectors basic to the rest of the paper:

*Definition 2:*

Be *Obj*, *Type* as defined in Def. 1. A type vector $T \subseteq (Type \cup (Obj \times Type))^n$ is a vector whose $i$-th element $T_i$ is the set of

- types to be contained by $obj_i$, and
- pairs of references to other objects' contained types and type constraints to be applied when including these in $T_i$.

Now, the assignment $obj_j^s \leftarrow obj_i^c$ will yield the type set $T_j = \{(obj_i, C_j)\}$. The meaning is that all types in $T_i$ of $Obj_i$ which meet constraint $C_j$ are to be included into $T_j$ when $T_i$ comes to be known: $T_j = T_j \cup (T_i \cap C_j)$. This is the same operation as performed by FPI (see Eq. 5) and will therefore be delegated to the FPI algorithm.

Another assignment, $obj_k^s \leftarrow obj_j^c$, may pass on the reference contained in $T_j$ to $T_k$. We have to consider that $obj_k$ does not receive types from $obj_i$ through a direct assignment, but indirectly via $obj_j$. Thus, constraint $C_k$ of the target *and* constraint $C_j$ of the intermediate object are applied: $T_k = \{(obj_i, C_j \cap C_k)\}$.

We now consider a concurrent object as a target of an assignment $obj_j^c \leftarrow obj_i$ from any source object in the same way as devised for sequential targets in the two previous paragraphs. If the source is concurrent, the pair $(obj_i, C_j)$ will represent element $A_{ij}$ of the assignment matrix (Eq. 6) in subsequent FPI. Since entries with $C_j = \varnothing$ wouldn't have any effect, they can be discarded. If the source is sequential, the link to $obj_i$ will result in the inclusion of DFA results for $obj_i$ into FPI. Also the propagation of links according to the previous paragraph will be performed with the side effect of faster FPI convergence due to the use of direct links to represent indirect data flow.

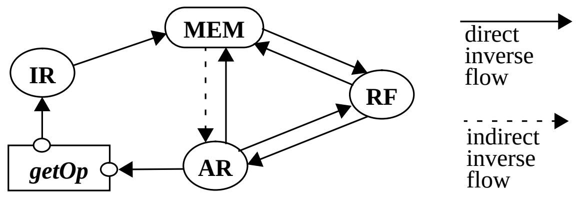


**Fig. 4: Type set links between polymorphic objects**

Fig. 4 shows how the type sets of objects will be linked to represent data flow of Fig. 2 and Ex. 2. Note that links are in the inverse direction of data flow. Solid lines represent direct assignments. The dotted line stands for a propagated link originating from AR receiving data from MEM via RF. There is, on the other hand, no such link from IR to RF because their type constraints are disjoint ($T_{IR} \cap T_{RF} = \varnothing$). Finally, the box hides function *getOp* called in Ex. 2. Handling of such hierarchy will be explained in section 5.3.

## 4 Resource allocation

Given objects *Obj* and type vector *T*, resource allocation means to determine and allocate for each single polymorphic object $obj_i \in Obj$ the minimum storage space required to contain a value of any type in $T_i$. The number of bits for $obj_i$ is computed by function *Bitsize* as follows:

- $obj_i$ must be capable of containing the largest type and a tag representing the current type of the object:

(8) $$Bitsize(obj_i) = max\{Bitsize(t) \mid t \in T_i\} + Tagsize(T_i)$$

- .The size of the tag depends on its encoding. For one-hot and fully encoded tags see Eq. 9. To save space, one could alternatively apply variable-length (Huffman) encoding, using less bits for the tag of types *t* with large *Bitsize*(*t*).

(9) $$Tagsize(T_i) = \begin{cases} card(T_i) & if\ \ one-hot\ \ encoded \\ \lceil ld(card(T_i)) \rceil & if\ \ fully\ \ encoded \end{cases}$$

- The size of a class type is the sum over the size of its attributes *a*. If the attribute is of a class type or polymorphic, *Bitsize*(*a*) will be computed recursively. Size of all primitive types is known.

(10) $$Bitsize(t) = \sum_{a \in Attributes(t)} Bitsize(a)$$

During synthesis, we will allocate a bitvector of *Bitsize*($obj_i$) bits to hold the state of the polymorphic object $obj_i$.

## 5 Implementation for Objective VHDL

In the previous sections, we have formulated the developed concepts for data type analysis universally so as to be applicable to object-oriented hardware models in general. We will now verify their practical applicability with an implementation for Objective VHDL, an extension of VHDL for, among other concepts, object-oriented class types, class inheritance, and polymorphism. The implementation will be part of a preprocessing system for the translation of Objective VHDL into plain VHDL, allowing to use existing VHDL tools for simulation and synthesis (Fig. 5). The system consists of the commercial LEDA front-end extended to parse and analyze Objective VHDL semantically, and the OFFIS back-end for optimization and Standard VHDL code generation.

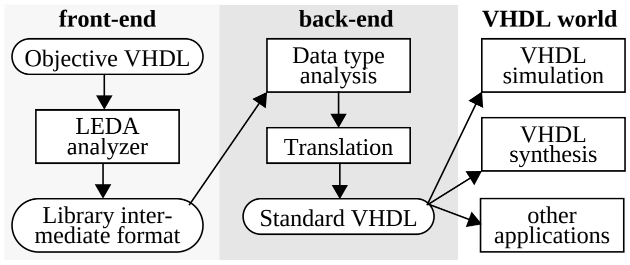


**Fig. 5: Objective VHDL preprocessor system**

In the following, we will report on the implementation of data type analysis for Objective VHDL. Data flow analysis as presented in 3.1 will be implemented with an optimized data structure and extended to cope with special VHDL features, namely loops with **exit** and **next** statements and hierarchical modeling. Fixed point iteration will be formulated as an algorithm on the devised data structure. Resource allocation in the sense of section 4 is a relatively straight-forward task within translation (cf. Fig. 5) and will not be considered here.

### 5.1 Basic data flow analysis

We represent the type vector *T* as defined in Def. 2 by a *declaration table* containing one entry for each polymorphic Objective VHDL variable or signal (object $obj_i$). Currently, the table is simply a linked list; it may however be implemented as a hash table to allow faster search for a particular object. The set of types and links $T_i$ corresponding to $obj_i$ is a linked list of elements as shown in Fig. 6. Elements contain either a reference to a class type or a link to another object in the declaration table together with the applicable type constraint. Such constraint is always a set including a parent class and its derived classes and can therefore be represented by a simple reference to the parent class.

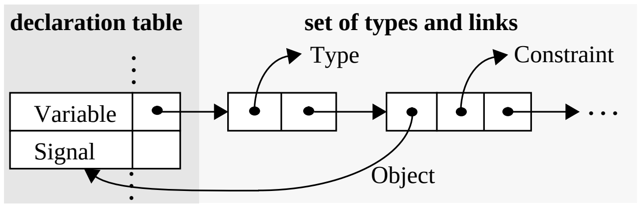


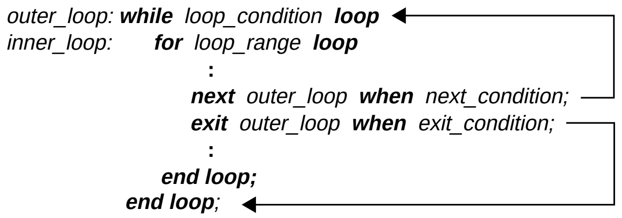
**Fig. 6: Data structure for type vector *T***

To implement the DFA equations of 3.1, the required operations, unification and intersection, are implemented on the devised data structure. DFA is carried out with a syntax-directed algorithm: For each kind of statement (assignment, sequence, alternative, loop) there is a procedure computing on the declaration table the corresponding transformation of type vector *T*. These procedures are invoked according to the sequence of statements in the Objective VHDL model under analysis.

### 5.2 Handling of loops

The problem with VHDL loops is that **exit** and **next** statements may be used to leave them and to jump back to their beginning, respectively. In nested loops, these statements can even jump across loop boundaries (see Ex. 3). We had to extend DFA techniques so as to cope with these features.

*Example 3:* **exit** and **next** statements in (Objective) VHDL loops



```
outer_loop: while loop_condition loop
inner_loop:      for loop_range loop
                        :
                     next outer_loop when next_condition;
                     exit outer_loop when exit_condition;
                        :
                 end loop;
             end loop;
```

The effect of a **next** statement is that not only the type vector $S1(\varnothing)$ generated by the complete interior of the loop (cf. Eq. 4) may be propagated to the beginning of the loop, but also the type vector $S1_{next}(\varnothing)$ generated up to the point before the **next** statement. An **exit** statement may cause the type vector valid before the **exit** to be propagated to the end of the loop. Considering multiple **next** statements, *next*(1) ... *next*(*m*), and multiple **exit** statements, *exit*(1) ... *exit*(*n*), we therefore re-write Eq. 4 as:

$$(11)\quad T^{out} = S(T^{in}) = \bigcup_{i=0}^{n} S1_{exit(i)}\left(T^{in} \bigcup_{i=0}^{m} S1_{next(i)}(\varnothing)\right),$$

where *next*(0) and *exit*(0) denote normal loop iteration and conclusion, respectively, at the end of the loop. Eq. 11 can be computed efficiently with two runs through *S*1.

A similar technique is used to handle **return** statements in procedures and functions. We also remark that **process** statements, which are restarted after their last statement, are handled like loops.

## 5.3 Handling of hierarchy

VHDL offers two sorts of hierarchical modeling to the designer[1]. First, design hierarchy may be described through instantiation of design entities (entity/architecture pairs). Second, functional decomposition into subprograms calling other subprograms is provided. With each call to a subprogram or instantiation of a design entity, different actuals may be passed to the formal parameters or ports, respectively. We take this fact into account through an *instance tree* (Fig. 7) corresponding to instantiation and call hierarchies. Each node (instance) has its own declaration table so that different instances can be analyzed independently.

The connection between an instance *I* and the higher level in hierarchy, *H*, is through interface objects (i.e., formal subprogram parameters and entity ports). If such interface object (see the bubbles of *getOp* in Fig. 4) is an input parameter or port, it gets a link to the assigned actual in the declaration table. If the interface object is for output, the link is in the other direction, and in case of inout objects there is one link in either direction. Thereby, connection between the instances is established. The flow of data types through the interface objects will be computed by FPI (see 5.4).

The case of recursive subprogram calls is irrelevant as not synthesizable with current tools. It could, however, be dealt with allowing loops in an instance graph and handling all the recursive invocations the same. Global signals declared in packages are taken into account through a list of used packages at the root of the instance tree.

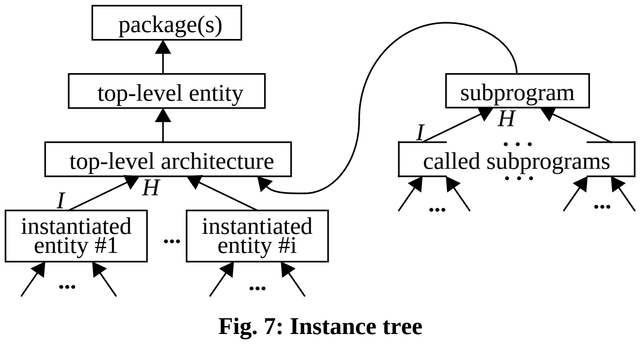


**Fig. 7: Instance tree**

## 5.4 Fixed point iteration

The implemented fixed point iteration algorithm listed in Fig. 8 computes the basic FPI equation (Eq. 5) and consideres hierarchy (cf. 5.3). It combines local iteration of a declaration table *T* and recursive descent into the instance tree.

The body of procedure *FPI* is repeated until all sets of types have converged (line 2). For each object $obj_j$ in *T* (line3) and all its links to objects $obj_i$ from which data are received (line 4), types are propagated from $obj_i$ to $obj_j$ according to Eq. 5 (line 8). Before, if $obj_i$ is located at a deeper level of hierarchy (line 5), the declaration table of the instance containing $obj_i$ is iterated (line 6) by a recursive call to *FPI*. In this case, it is first checked by call to a function, *SteadyState*, in line 2 whether the set of types for an input of that instance has changed. If not, *FPI* can return immediately; otherwise iteration must be carried out.

```
(1)   procedure FPI( T ) is
(2)       while not SteadyState( T ) loop
(3)           for j in 1 to n loop
(4)               for each ( $obj_i$ , $C_j$ ) in $obj_j$ loop
(5)                   if DeeperHierarchy( $obj_i$ )
(6)                       FPI( DeclTable( $obj_i$ ) );
(7)                   end if;
(8)                   $T_j := T_j \cup ( T_i \cap C_j )$;
(9)               end loop;
(10)          end loop;
(11)      end loop;
(12)  end;
```

**Fig. 8: Fixed-point iteration algorithm**

## 6 Results

The implemented analysis tool has been applied to a model of the microprocessor data flow (Fig. 2). Results are listed in Table 1. An 'X' means that a polymorphic object (rows) must be able to contain a value of a class type (column). A '–' stands for a class type which need not be considered thanks to data type analysis results. The blank fields result from types which are incompatible with the type constraint of the object. Finally, bit-size values shown in the right-hand columns have been calculated from the analysis results according to section 4. Values correspond to automatic optimization using methods presented in this paper (auto), manual implementation of the 32-bit processor with bitvector data types (man), and synthesis without any optimization (nopt).

| | Byte | Word | Reg Add. | Mem Add. | Instr. | auto [bit] | man [bit] | nopt [bit] |
|---|---|---|---|---|---|---|---|---|
| MEM | X | X | — | X | X | 34 | 32 | 80 |
| IR | | | | | X | 32 | 32 | 80 |
| IR.OP | X | — | X | — | | 9 | 8 | 32 |
| RF | X | X | — | X | | 34 | 32 | 34 |
| AR | | | X | X | | 33 | 32 | 33 |

**Table 1: Optimization results**

The main optimization potential in this case stems from instructions having only short operands (IR.OP) of class types Byte and RegAddr, but no full 32-bit words or memory addresses. This has been correctly recognized by analysis, allowing the reduction of IR and MEM width to 32 and 34 bits, respectively, instead of an unoptimized 80 bits. However, each memory element still has 2 bits more than a manually implemented 32-bit architecture. The reason are the tags used to distinguish values of the four class

1. Objective VHDL adds a third kind, hierarchical modeling of data with associated functionality through an inheritance hierarchy.

types present in memory, requiring 2 bits when fully encoded (cf. section 4). While these tags are a nice feature to detect run-time errors during simulation, e.g. loading of an operand into the instruction register, a manually designed hardware implementation of the microprocessor would in some cases make no difference between the encoded types. For instance, an operand would be re-interpreted as instruction when loaded into IR. On the other hand, the different instruction types definitely must be told apart by a tag (i.e. opcode). We are currently implementing techniques to automatically determine from control and data flow cases when a tag can be omitted so as to automatically accomplish the bit-widths that would result from manual design.

To show the practical applicability of data type analysis, we have measured run time of our implementation. Table 2 lists CPU time required on a Sparc 20 workstation for data flow analysis and fixed point iteration of ten benchmarks. The table also shows for each benchmark its numbers of component instances, polymorphic objects (50% of which are signals and 50% variables), assignment operations between these objects, and interface objects through which instances communicate. Benchmarks are worst case to analysis run time in the sense that each of the 50 used class types is propagated through the complete hierarchy with the result that each polymorphic object has to hold the maximum set of types (i.e. all types allowed by its type constraint).

| | instances | polym. objects | assignments | interf. objects | DFA [s] | FPI [s] |
|---|---|---|---|---|---|---|
| a1 | 1 | 10 | 32 | 0 | 0.05 | 0.02 |
| a2 | 3 | 30 | 96 | 4 | 0.07 | 0.12 |
| a3 | 7 | 70 | 224 | 12 | 0.12 | 0.57 |
| a4 | 15 | 150 | 480 | 28 | 0.18 | 2.00 |
| a5 | 31 | 310 | 992 | 60 | 0.35 | 6.02 |
| a6 | 63 | 630 | 2016 | 124 | 0.70 | 17.05 |
| a7 | 127 | 1270 | 4064 | 252 | 1.37 | 45.48 |
| a8 | 255 | 2550 | 8160 | 508 | 2.80 | 118.02 |
| a9 | 511 | 5110 | 16352 | 1020 | 6.10 | 299.55 |
| a10 | 1023 | 10230 | 32736 | 2044 | 13.15 | 731.63 |

**Table 2: Optimization run times (worst case)**

With each row, the number of polymorphic objects and their assignments, $n$, approximately doubles. While DFA run time doubles as well, the more significant FPI run time tends to triple. Thus, we estimate an experimentally obtained complexity O( $1.5n$ ) when design hierarchy and limited communication through interface objects can be exploited, as opposed to the analytical worst-case O( $n^3$ ) for non-hierarchical models. Since a model with 10230 instances of complex data types would result in a large hardware implementation (130k gates only for registers when assuming 32-bit objects), the 12 minutes required for its optimization are justified. If a model should be too large to be handled fast enough by our implementation, we could still split it into separately analyzed parts.

## 7 Conclusion

As part of our work on synthesis from object-oriented hardware models, we have presented data type analysis techniques targeting at an optimized synthesis of class types and polymorphism. Analysis is carried out statically with a combination of data flow analysis techniques for objects accessed sequentially and a fixed-point iteration for objects in the concurrent domain. Its results can be used to implement objects with a minimized number of bits.

We have implemented our analysis methods for Objective VHDL, an object-oriented extension to VHDL, and reported experimental optimization and run time results. Yet, concepts have been formulated general enough to be helpful also for synthesis from other object-oriented languages.

Experimental results show a significant benefit of our techniques as compared to non-optimized synthesis of object-oriented data types. Currently, we cannot fully achieve the results one would obtain from a manual design at "bit level", i.e. without data abstraction. However, our future work includes the implementation of techniques to utilize further optimization potential.

From a run time performance perspective, the available implementation of data type analysis can be applied to considerably large hierarchical models. There are starting-points to further optimize the software, and even if a model would turn out too large for global analysis, one could still split it to locally optimize the parts.

## 8 References

[1] A. Allara, M. Bombana, P. Cavalloro, W. Nebel, W. Putzke-Röming, M. Radetzki. *ATM cell modelling using Objective VHDL*. Proc. Asia South Pacific Design Automation Conference (ASP-DAC), 1998, pp. 261-264.

[2] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] P. J. Ashenden, P. A. Wilsey, D. E. Martin. *SUAVE: Painless Extension for an Object-Oriented VHDL*. Proc. VHDL Int'l Users' Forum (VIUF, Fall Conference), 1997, pp. 60-67.

[4] M. Aubury, I. Page, G. Randall, J. Saul, R. Watts. *Handel-C Language Reference Guide*. Technical Report, Oxford University Computing Laboratory, 1996.

[5] G. Booch. *Object Oriented Design*. Benjamin/Cummings Publishing, Redwood City, 1991.

[6] S.-T. Cheng, P. C. McGeer, M. Meyer, T. Truman, A. Sangiovanni-Vincentelli, P. Scaglia. *The V++ System Design Language*. Proc. Design Automation and Test in Europe (DATE), Designer Track, Paris, France, 1998, pp. 3-10.

[7] R. Helaihel, K. Olukotun. *Java as a Specification Language for Hardware-Software-Systems*. Proc. IEEE/ACM Int'l Conf. on Computer Aided Design (ICCAD'97), San Jose, California, 1997, pp. 690-697.

[8] E. Holz et al. *INSYDE Integrated Methods for Evolving System Design—Application Guidelines*. ESPRIT Project 8641 Report, Humboldt University Berlin.

[9] *IEEE Standard VHDL Language Reference Manual*. IEEE Std 1076-1993.

[10] S. Kumar, J. H. Aylor, B. W. Johnson, W. .A. Wulf. *The Codesign of Embedded Systems: A Unified Hardware/Software Representation*. Kluwer, 1996.

[11] M. Radetzki, W. Putzke-Röming, W. Nebel, S. Maginot, J.-M. Bergé, A.-M. Tagant. *VHDL language extensions to support abstraction and re-use*. Proc. 2nd Workshop on Libraries, Component Modelling, and Quality Assurance. Toledo, Spain, 1997, pp. 47-62.

[12] G. Schumacher, W. Nebel. *Inheritance Concept for Signals in Object-Oriented Extensions to VHDL*. Proc. Euro-DAC with Euro-VHDL, IEEE CS Press, 1995, pp. 428-435.

[13] S. Swamy, A. Molin, B. Covnot. *OO-VHDL. Object-Oriented Extensions to VHDL*. IEEE Computer, October 1995.