Synthesis of Object-Oriented Descriptions Modeled at Functional-Level

MAZIAR GOUDARZI, SHAAHIN HESSABI
Department of Computer Engineering
Sharif University of Technology
Tehran, I.R.IRAN

Abstract: In this paper we present a novel approach to synthesizing polymorphic hardware from functional-level Object-Oriented (OO) models. Our proposed target architecture is intended to be subsequently used in co-design of hardware and software from a single OO source model, and hence, in order to be consistent with software compilers uses a global memory to store the objects data. The target architecture enables inheritance and polymorphism; it consists of a method-invocation unit (MIU), a functional unit (FU) per class method, and an object-management unit (OMU). The MIU, FU, and OMU functionalities are presented, along with the details of modeling and synthesis of a traffic-light controller to demonstrate the concepts and approach. Moreover, experimental results of modeling and implementing some case studies are presented and analyzed. Synthesis results show some area overhead compared to traditional methodologies; however, we show that this overhead is constant for all applications having the same class hierarchy and moreover is independent of the number of objects in the system. Therefore this overhead is ignorable, or even not present, for large designs or designs with several objects where OO shows its best.

Key-Words: hardware synthesis, object-oriented synthesis, synthesis of polymorphism, hardware/software co-design, functional-level object-oriented modeling

1 Introduction

SIA roadmap for semiconductors design [1] shows that for the last two decades potential design complexity has grown at a rate of 58% per year, while the designer productivity at the same time has only raised 21% per year. This has led to a growing design productivity gap between manufacturing capability of chips and the functionality that designers can implement in unit time. The manufacturing capability is predicted to grow at the same rate for another decade and hence the gap must be filled by increasing designer productivity rate of growth.

Currently, hardware design methodologies mainly focus on structural decomposition (i.e. hierarchical design) to overcome design complexity. Reuse of IP cores (as in core-based or IP-based design [2]) and of target architectures (as in platform-based design [3]) are other points of focus to raise the designer productivity. But the gap still exists.

To fill in the gap, new design methodologies and tools are quite necessary. Object-oriented (OO) design, successfully used for several years by the software community, is a rather different approach to complexity management compared to traditional hardware design methodologies. Traditional methodologies, force the designer to more or less imagine the final architecture even when designing the high-level models. In OO methodology, however, the designer first distinguishes the main data types (or classes) present in the system and the operations that should be applied to them [4]. More consistent to the natural world, the whole system is composed of data (or objects) of those types which are interacting by calling one another’s methods. In other words, OO methodology suggests modeling the system in terms of its constituting data objects, while traditional methodologies concentrate on structurally decomposing the target architecture of the system.

OO can be applied to hardware design at various levels of abstraction. Most researchers in this field have applied OO semantics at module- or entity-level [5] [6] or ultimately view and implement the objects as structural hardware elements [7]; consequently, they lead to either inability or inefficient implementation of important OO features such as inheritance and polymorphism. In such approaches, classes directly correspond to hardware modules; i.e., objects in the model are
structural objects. Although such views are useful and can contribute to closing the productivity gap by allowing (if at all worked on) class hierarchy and inheritance, we believe that there is a better approach. OO modeling can be much more useful when applied at the highest possible levels of abstraction; we believe that the user should model his system at functional-level, totally forgetting that it will be implemented in hardware. Then during the refinement steps, appropriate EDA tools should extract necessary information out of the model and synthesize it in hardware. This is a true top-down design that increases designer productivity by hiding all unnecessary implementation details, in addition to re-use and flexibility advantages resulting from inheritance and class hierarchy. We wish to view the system as a composition of interacting functional objects whose behaviors (class methods) are modeled at behavioral level and synthesized using currently mature behavioral synthesis techniques. Hence, we believe that OO modeling is the next step over behavioral modeling towards higher abstraction levels. The detailed OO model of a traffic-light controller presented in the next section reveals our view of functional-level OO modeling.

Although OO hardware modeling and simulation are quite useful, it is unlikely to be widely used unless a path to synthesis is also provided. Moreover, inheritance and polymorphism are key to major features and advantages of OO design and hence supporting them is crucial for the synthesis solution. Our proposed target architecture, effectively an adaptation of the software OO compilation approach to hardware synthesis, implements them by synthesizing an environment to dynamically (at run-time) determine the actual method to call.

We intentionally ignore the specification, simulation, and synthesis languages and focus on the target architecture and mapping OO constructs to it, so as to motivate and enable OO design regardless of the language. The case studies’ initial models are presented in pseudo-C++ format and the simulation and synthesis models are in Verilog®. As Verilog® lacks required OO constructs (class and object definition and object-method calls) they are done in paper and pencil and only the final implementation is presented. This, however, shows that an appropriate EDA tool can provide the designer with necessary facilities to define the classes, instantiate objects, call object methods, and use Verilog® (or any desired HDL) merely to define the class methods.

In the rest of this paper, in Section II we present a case study of OO modeling at functional-level to clarify and illustrate the concept. Meanwhile, the underlying scheduling model is introduced. Section III presents our approach to OO synthesis, the proposed target architecture, and details of how it implements inheritance and polymorphism. Section IV presents details and procedure of synthesizing the model in Section II. Experimental results of synthesizing that model along with its traditional model and two other OO models are presented and analyzed in Section V. Finally, Section VI summarizes and concludes the paper.

2 Object-Oriented Modeling at Functional-Level

In this section, the classic example of traffic light controller is adopted to show the advantages of OO modeling at functional level, while introducing our model for operation scheduling in the target architecture. As a summary of the problem, a controller is to be designed for the crossroad of a highway and a farm-road. The highway should always remain open unless a car appears at the farm-road. In this situation, if the highway has been open for at least a $\text{min\_green}$ time, it is closed and the farm-road temporarily opened for a fixed $\text{fixed\_green}$ time after which the highway is re-opened. Examples of traditionally modeling this problem can be found at many digital design references including [8]. For the OO model, we use pseudo-C++ codes for the sake of clarity and easier understanding. Our proposed synthesis in Verilog® is presented in Section IV.

Investigating the problem, one can figure out that the main data types (classes) in the problem are the two traffic light types; i.e., the highway and the farm-road lights. The class hierarchy in Fig. 1 can be proposed rooted at $\text{traffic\_light}$ class. This class is an abstraction of traffic lights and hence covers only their common features. This includes a $\text{state}$ (whether light is in red, green, or yellow state) and a $\text{elapsed\_time}$ (how long this state has been active) as data members, and three methods $\text{open()}$, $\text{close()}$, and $\text{timekeeper()}$, which turn the light to green, turn it to red, and keep track of time units, respectively.
In our problem, there are two variants of (or derivations from) this class: farm-road and highway lights. Hence, two classes are derived: farmroad_light and highway_light. The former adds a new data member named fixed_green and specializes open() method according to its definition. The latter adds the new min_green data member and overrides close() method. Details of the class hierarchy and methods are presented in Fig. 2. The update_state() function sets the state member to the new value and resets elapsed_time.

```
1 main0() { 
  2 frl.close();
  3 hw1.open();
  4 forever do {
    5 if(frl_sensor){
      6 hw1.close();
      7 frl.open();
      8 frl.close(); // unnecessary
      9 hw1.open();
    10 } } // forever
  11 } //main0
12 } //main1
```

This simple example shows that OO modeling clearly separates functionality (objects definitions in classes) from communication (objects interactions in main() functions or methods) which is very advantageous in

```
1 main0() {
  2 frl.close();
  3 hw1.open();
  4 forever do {
    5 if(frl_sensor){
      6 hw1.close();
      7 frl.open();
      8 frl.close(); // unnecessary
      9 hw1.open();
    10 } } //forever
  11 } //main0
12 } //main1
```

Fig. 3. Object instantiations, and the two main functions

Compiling (or manually investigating) the two main() functions, a directed flow graph, called Method Flow Graph (MFG), can be extracted that shows the sequence, parallelism, and conditions of object.method calls at run-time. Each node (state) represents an object.method pair to call, which itself can possess another MFG. Each (possibly tagged) edge represents flow of control when the method at the source node is finished and the tag condition is true. "nop" states are inserted where no method is to be called but some other operation should execute or a condition checked. This highly resembles a Hierarchical Concurrent Finite State Machine (HCFSM) [9] representing entire system state in terms of methods to activate. MFGs of the main0() and main1() functions are shown in Fig. 4. In our example, the nodes are not hierarchical.

Note the difference between functional and structural objects in the above example. The hw1 and frl objects need not necessarily correspond directly to hardware modules. They are functional objects which can, as presented in next section, be realized in hardware but not directly to a certain module.

This simple example shows that OO modeling clearly separates functionality (objects definitions in classes) from communication (objects interactions in main() functions or methods) which is very advantageous in
managing complexity of designs. As can be seen from
the example, the crossroad operation policy (i.e. the
interaction or communication among light objects) is
effectively separated from the lights functionality. As
will be presented shortly below, the same traffic_light
class can be reused in many other similar scenarios;
e.g., a pedestrian light can be added, or support for
emergency vehicles included. The designer only needs
to derive a new class and write necessary methods. The
crossroad policy of operation can also be easily
changed by merely modifying the main0() function.
Such changes are very hard to apply in a traditional
modeling approach.

3 The Synthesis Approach

In the first subsection, the way software compilers
implement inheritance and polymorphism is briefly
presented. Our approach is an adaptation of the same
technique to the hardware world. Its mechanism of op-
eration and the way it enables polymorphism is pre-
sented in subsection B. The two fundamental modules,
the Method-Invocation Unit (MIU) and the Object-
Management Unit (OMU), are detailed next.

3.1 Software Compilation of OO Models

In a class hierarchy, every child class inherits all data
and methods from its parent(s). The child class is al-
lowed to add (but not remove) any required data and
methods. Besides, it can redefine (override) any inher-
ited method to accommodate it to its own specific re-
quirements. At run time, the class of the calling object
determines which class method (of the parent or child)
will actually get called. Hence, calling the same method
on different objects results in different behaviors. This
phenomenon is known as polymorphism, which allows
the same calling code treat different objects differently
(and accordingly of course).

Compilers of OO software, like Java or C++ compil-
ers, use the following method in compilation [10]. They
store objects data in the processor memory space in a
contiguous block for each single object. The object’s
type (class) is stored at the same block as an implicit
data member. All the class methods are compiled as
ordinary functions having their declared arguments plus
a compiler-inserted pointer argument named this*, that
at run-time points to the location in memory where the
object data is stored. This allows the same code to op-
erate on all objects of the same class (or classes derived
from it). Besides, for each class a Virtual Method Table
is generated to store pointers to the compiled code of
class methods. When the object type is not known at
compile-time (e.g. when using an object pointer to
specify the object), calling an object’s method is done
at run-time by detecting object type from object’s stor-
age block, and looking up the address of the desired
method from the class’s VMT. Therefore, the actual
method to call can vary at run-time according to the
object, and hence, polymorphism is realized.

3.2 Realizing Hardware Polymorphism

We are adapting the above software approach to the
specifics of hardware. The work is to be extended to a
fully-OO system-level design methodology and envi-
ronment. In the authors’ viewpoint, the final implemen-
tation in such an environment would be a set of soft-
ware-implemented class methods that along with an-
other set of hardware-implemented ones operate on a
shared set of objects stored in a global shared memory.
OO modeling has been applied to software for many
years and OO compilers are readily available for vari-
ous processors. Hence, it would be wise to remain con-
istent with them in the key point (the way objects are
stored in memory) and devise the hardware synthesis
method accordingly. So, the objects data are assumed to
be stored in a memory shared among processor(s) and
hardware block(s).

Fig. 5 shows, in an overall view, the elements of our
proposed architecture. As an example, two classes,
namely A and B, are defined in the system along with
three objects. Objects O_A1 and O_A2 are of class A, and
object O_B1 is of class B. Class A has defined f1() and
f2() methods. Class B, derived from A, has added f3() method and overridden f1().

As in software compilation, each object’s data is
stored in a contiguous block, as shown in “Memory”
box in Fig. 5. Any B object owns the same data ele-
ments as A objects, appended with some B-specific
data shown in gray there. The memory is accessible to
the hardware block over a bus. The example model has
four class methods, named A.f1(), A.f2(), B.f1(), and
B.f3(), all of which are implemented in functional units
(FU) shown in dark gray boxes in the HW block rec-
tangle. A Method-Invocation Unit controls the coarse-
grained sequence and parallelism of operations in the
system by activating appropriate FU(s). Each FU con-
trols its own internal operations viewed as fine-grained
control of the system. From the MFG of the input
model, detailed in next section, the MIU knows which
object.method to call at each specific point in time. Using two tables, this pair is mapped at run-time to the appropriate FU to activate, which is where polymorphism (virtual-method dispatch) is realized. To access data of the called objects, the MIU passes the Object Identifier (oid) to the FU. The oid performs the same role as *this* in software. This oid concatenated with the index of the desired data of the object constitute a virtual address which is passed to an Object Management Unit (OMU) to access object’s data in memory. The MIU itself may also need to access some object or system data, and hence, has the same interface to the OMU.

The architecture enables polymorphism by allowing the class method (and hence the FU) to be determined at run-time. The MIU maps the object.method into appropriate FU, the FU operates transparently on the object determined by the oid, and the OMU maps the virtual to physical address to allow relocation of objects data and obtain software compatibility. The next two subsections provide details of the MIU and OMU. Section IV presents how the specification of MIU, OMU, and FUs are extracted from the input OO model.

### 3.3 The Method-Invocation Unit

The central role in realizing polymorphism, i.e. mapping object.method to FU, is played by this module. It is also responsible for controlling schedule of FU activations as the coarse-grained control unit. Each FU is activated by asserting the start signal. When the FU accomplishes its task, it informs the MIU by asserting done signal.

As described in Section II, an MFG can be extracted for the top-level thread(s) of execution and each class method. Each node in MFG shows its corresponding object.method by an (oid, mid) pair, representing the Object Identifier (oid) and the Method Identifier (mid). The corresponding class of the object is determined from Objects Type Table (OTT) that can be generated automatically at compile-time. Although this could be read from objects storage in memory, as in software, we propose synthesis of this table to reduce both memory load and scheduling delay overhead. The looked up Class Identifier (cid) together with the mid are mapped to the appropriate FU using the VMT. The VMT is also compile-time generatable due to availability of classes and declaration of their methods, as in software. Dynamic (run-time) object instantiation will be possible if OTT is allowed to dynamically grow. In summary, the above mapping can be expressed as:

\[(oid, mid) \xrightarrow{OTT} (cid, mid) \xrightarrow{VMT} FU\]

The scheduling duty of the MIU is accomplished according to the MFG. A set of concurrent finite-state machines is extracted from the top-level MFG, which designates the sequence and parallelism of operations. These state-machines can statically be hardwired in the MIU, or micro-programmed to allow control-flow alteration at run-time or after manufacturing. These two implementation styles will be illustrated shortly when synthesizing the case study.

### 3.4 The Object Management Unit (OMU)

Each FU implements a class method. Naturally, it needs to access object’s data to accomplish its task. This is done through OMU. The FU produces a virtual address consisting of the oid of the object on which it is working, concatenated by the index of its desired

---

**Fig. 5. Overall block-diagram view of the target architecture**
data in the object’s storage block: virtual_addr={oid,index}. OMU receives this and maps it to the data physical address using its internal table(s). The mapping tables are determined at compile-time when the objects data are laid out in memory. Data and control signals connect the FUs to the OMU to enable read-write operations.

One point specific to the OO hardware modeling is the presence of physical ports that connect the hardware block to the external world. These are not normally considered in software OO modeling but cannot be ignored in hardware. In our approach, a distinction is made between normal object data and the physical IO of the objects and system. This is reflected in the virtual address by a single bit field. OMU detects this and maps it accordingly. Based on application requirements or designer’s choice, the physical IO can be realized as isolated IO (lower part in Fig. 5), memory-mapped IO (left part in Fig. 5), or even direct connection of the physical ports and signals to the OMU.

As Fig. 5 shows, multiple FUs may request data access from OMU at the same time, which may be for different or the same data element. For non-conflicting data, requests need only be serialized. For conflicting access requests and synchronization purposes, OMU provides locking mechanism to implement atomic read-modify-write cycles as a basic mechanism. This is done by a lock signal during read-write cycles which prohibits other FUs from accessing the data element until the unlock operation is performed by the locking FU. It is up to the designer to use this mechanism when appropriate. The OMU is not responsible to take care of conflicting data access.

As the OMU functionality is generally independent from the input OO model, any other concurrent data access management policy can be implemented and/or chosen by the designer at compile-time. Besides, OMU is transparent to the FUs, and hence, can implement a caching policy or any other technique used in processor-based systems to improve memory performance.

4 Synthesis Procedure

The case study presented in Section II is manually synthesized into Verilog® HDL. Although both invariant and application-dependent portions of the target architecture are synthesized manually, those parts that can be automatically generated (i.e., module interfaces and wirings of the target architecture, MFGs extractable from objects interaction, and the mapping table of the OMU and MIU) are mentioned accordingly to motivate automating them in an EDA tool.

As described in Section II, five class methods exist in this system: open(), close(), and timekeeper() from traffic_light class, open() for farmroad_light class, and close() for highway_light class. To clarify the FU synthesis procedure, the Verilog code of open() method of the traffic_light class, or the traffic_light$open FU, is shown in Fig. 6. The bold italic portions of the figure provide an FU- and application-invariant template merely required to define the target architecture and are automatically generatable. These include declaration and definition of ports, declaration of constant parameters such as data width and oid width, and declaration of some tasks to access object data through OMU. All other FUs are defined in the same template. The designer only needs to specify the method functionality using defined tasks to access objects data elements. The method functionality is specified in normal font in the right panel of Fig. 6. Comparing it to the pseudo-C++ code of Fig. 2 shows that even the initial model can be in behavioral Verilog®.

There are two threads of control in this interesting case study. The second thread consists of calls to timekeeper() method for the two objects. The timekeeper() method is shown in Fig. 7 to show how locking mechanism (statements in italic font) is used in concurrent object data access.
The MIU operation is specified by concurrent FSMs extracted out of system MFG. Fig. 4 presented our case study MFG which can also be viewed as two FSMs with exactly the same structure. These two are implemented in our MIU module, summarized in Fig. 8.

All of the MIU code can be generated automatically; and hence, is shown in bold font. Fig. 8 shows both micro-programmed (first always construct, lines 15 to 28) and hardwired (second always construct, lines 30 to 39) implementation styles for threads of execution (details follows). The major advantage of the micro-programmed approach is its post-manufacturing programmability; this allows the same chip to be reused for other similar applications modeled with the same class hierarchy. This programmability is crucial nowadays when programmable chips are replacing ASICs [11].

The first always construct of Fig. 8 implements main0() thread. In this always block, reset condition is handled first (line 16). Then at line 20, the (oid,mid) pair to call is determined from a table based on the current state; this is where micro-programming is implemented. The oid to cid mapping is done next by OTT table (line 22), and finally the (cid, mid) pair is mapped to proper FU by virtual method table VMT (line 23). The last two mappings facilitate polymorphism, although not required in our simple case study. The functionalities of activate_FU() and next_state() tasks (lines 25 and 26) follow their names.

The second always statement in Fig. 8 shows hardwired implementation style. The FU to activate and the oid of the object on which to operate is readily available by investigating main1(). Hence, the sequence of their activations can be hard-coded in Verilog®, as in lines 30 to 40.

In general, the OMU functionality is independent from the OO input model except for the virtual to physical mapping table; and hence, can be realized once, and reused multiple times. Several extensions can be applied to it, e.g. caching, dynamic priority for FU requests, etc. We implemented a simple fixed-priority one merely responding to object access requests.

The MIU module implementing traffic light controller MFG

![Module Traffic Light Timekeeper](image)

**Fig. 7.** The timekeeper() method of traffic_light class, showing in italic font the locking mechanism to manage concurrent data access.

**Fig. 8.** The MIU module implementing traffic light controller MFG

<table>
<thead>
<tr>
<th>Table 1. Comparison of synthesis results of OO and traditional models on a sample 0.5u ASIC process</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Table Image" /></td>
</tr>
</tbody>
</table>

The second always statement in Fig. 8 shows hardwired implementation style. The FU to activate and the oid of the object on which to operate is readily available by investigating main1(). Hence, the sequence of their activations can be hard-coded in Verilog®, as in lines 30 to 40.

In general, the OMU functionality is independent from the OO input model except for the virtual to physical mapping table; and hence, can be realized once, and reused multiple times. Several extensions can be applied to it, e.g. caching, dynamic priority for FU requests, etc. We implemented a simple fixed-priority one merely responding to object access requests.

The MIU module implementing traffic light controller MFG

![Module Traffic Light Timekeeper](image)

**Fig. 7.** The timekeeper() method of traffic_light class, showing in italic font the locking mechanism to manage concurrent data access.

**Fig. 8.** The MIU module implementing traffic light controller MFG

<table>
<thead>
<tr>
<th>Table 1. Comparison of synthesis results of OO and traditional models on a sample 0.5u ASIC process</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Table Image" /></td>
</tr>
</tbody>
</table>

The second always statement in Fig. 8 shows hardwired implementation style. The FU to activate and the oid of the object on which to operate is readily available by investigating main1(). Hence, the sequence of their activations can be hard-coded in Verilog®, as in lines 30 to 40.

In general, the OMU functionality is independent from the OO input model except for the virtual to physical mapping table; and hence, can be realized once, and reused multiple times. Several extensions can be applied to it, e.g. caching, dynamic priority for FU requests, etc. We implemented a simple fixed-priority one merely responding to object access requests.

The MIU module implementing traffic light controller MFG

![Module Traffic Light Timekeeper](image)

**Fig. 7.** The timekeeper() method of traffic_light class, showing in italic font the locking mechanism to manage concurrent data access.

**Fig. 8.** The MIU module implementing traffic light controller MFG

<table>
<thead>
<tr>
<th>Table 1. Comparison of synthesis results of OO and traditional models on a sample 0.5u ASIC process</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Table Image" /></td>
</tr>
</tbody>
</table>

The second always statement in Fig. 8 shows hardwired implementation style. The FU to activate and the oid of the object on which to operate is readily available by investigating main1(). Hence, the sequence of their activations can be hard-coded in Verilog®, as in lines 30 to 40.

In general, the OMU functionality is independent from the OO input model except for the virtual to physical mapping table; and hence, can be realized once, and reused multiple times. Several extensions can be applied to it, e.g. caching, dynamic priority for FU requests, etc. We implemented a simple fixed-priority one merely responding to object access requests.

The MIU module implementing traffic light controller MFG

![Module Traffic Light Timekeeper](image)

**Fig. 7.** The timekeeper() method of traffic_light class, showing in italic font the locking mechanism to manage concurrent data access.

**Fig. 8.** The MIU module implementing traffic light controller MFG

<table>
<thead>
<tr>
<th>Table 1. Comparison of synthesis results of OO and traditional models on a sample 0.5u ASIC process</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Table Image" /></td>
</tr>
</tbody>
</table>

The second always statement in Fig. 8 shows hardwired implementation style. The FU to activate and the oid of the object on which to operate is readily available by investigating main1(). Hence, the sequence of their activations can be hard-coded in Verilog®, as in lines 30 to 40.

In general, the OMU functionality is independent from the OO input model except for the virtual to physical mapping table; and hence, can be realized once, and reused multiple times. Several extensions can be applied to it, e.g. caching, dynamic priority for FU requests, etc. We implemented a simple fixed-priority one merely responding to object access requests.

The MIU module implementing traffic light controller MFG

![Module Traffic Light Timekeeper](image)

**Fig. 7.** The timekeeper() method of traffic_light class, showing in italic font the locking mechanism to manage concurrent data access.

**Fig. 8.** The MIU module implementing traffic light controller MFG

<table>
<thead>
<tr>
<th>Table 1. Comparison of synthesis results of OO and traditional models on a sample 0.5u ASIC process</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Table Image" /></td>
</tr>
</tbody>
</table>

The second always statement in Fig. 8 shows hardwired implementation style. The FU to activate and the oid of the object on which to operate is readily available by investigating main1(). Hence, the sequence of their activations can be hard-coded in Verilog®, as in lines 30 to 40.

In general, the OMU functionality is independent from the OO input model except for the virtual to physical mapping table; and hence, can be realized once, and reused multiple times. Several extensions can be applied to it, e.g. caching, dynamic priority for FU requests, etc. We implemented a simple fixed-priority one merely responding to object access requests.

The MIU module implementing traffic light controller MFG

![Module Traffic Light Timekeeper](image)

**Fig. 7.** The timekeeper() method of traffic_light class, showing in italic font the locking mechanism to manage concurrent data access.

**Fig. 8.** The MIU module implementing traffic light controller MFG

<table>
<thead>
<tr>
<th>Table 1. Comparison of synthesis results of OO and traditional models on a sample 0.5u ASIC process</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Table Image" /></td>
</tr>
</tbody>
</table>

The second always statement in Fig. 8 shows hardwired implementation style. The FU to activate and the oid of the object on which to operate is readily available by investigating main1(). Hence, the sequence of their activations can be hard-coded in Verilog®, as in lines 30 to 40.

In general, the OMU functionality is independent from the OO input model except for the virtual to physical mapping table; and hence, can be realized once, and reused multiple times. Several extensions can be applied to it, e.g. caching, dynamic priority for FU requests, etc. We implemented a simple fixed-priority one merely responding to object access requests.

The MIU module implementing traffic light controller MFG

![Module Traffic Light Timekeeper](image)

**Fig. 7.** The timekeeper() method of traffic_light class, showing in italic font the locking mechanism to manage concurrent data access.

**Fig. 8.** The MIU module implementing traffic light controller MFG

<table>
<thead>
<tr>
<th>Table 1. Comparison of synthesis results of OO and traditional models on a sample 0.5u ASIC process</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Table Image" /></td>
</tr>
</tbody>
</table>

The second always statement in Fig. 8 shows hardwired implementation style. The FU to activate and the oid of the object on which to operate is readily available by investigating main1(). Hence, the sequence of their activations can be hard-coded in Verilog®, as in lines 30 to 40.

In general, the OMU functionality is independent from the OO input model except for the virtual to physical mapping table; and hence, can be realized once, and reused multiple times. Several extensions can be applied to it, e.g. caching, dynamic priority for FU requests, etc. We implemented a simple fixed-priority one merely responding to object access requests.
5 Experimental Results

The whole model is simulated and validated using ModelSim® commercial HDL simulator. The initial model, as presented in Figures 6 to 8, was not directly synthesizable by LeonardoSpectrum® commercial synthesis tool. This was due to the use of timing constructs in the task definitions of the FUs and MIU, and due to concurrent access to module variables in the OMU specification. We inclined the tasks and integrated all of the concurrent accesses into a single always block to synthesize them. The area/delay/memory synthesis results are presented in Table 1.

Another traffic-light problem concerning the four lights installed at the crossing of four streets at a square was also modeled and synthesized using our proposed OO approach. Fig. 9 shows the object instantiations (Global objects window), the lights operation policy (First main window), and the time tracking thread (Second main window). These are, as a great advantage, the only code required for the new application to finish modeling. All class definitions are reused from the previous example, and hence, the modeling time is highly reduced. Note that no call to close() method is required in the forever loop due to its being embedded in the open() method of farmroad_light objects. The synthesis results are shown in the third row of Table 1. The same problem was modeled again, this time by objects of type traffic_light instead of farmroad_light. The new synthesis results are also shown in Table 1 at last row.

As Table 1 shows, the total area of the OO traffic light controller is much higher than the traditional model; however, it should be noted that we made no attempt to area-optimize our target architecture modules (specially the OMU that introduces the highest overhead and can be highly optimized by manual design, instead of current behavioral synthesis, since it is not application-dependant.) The other two examples also show the area overhead, although their corresponding traditional model is not presented. We would like to point out that this area overhead is independent of the number of objects in the system. For another similar application with several traffic lights, the area of the traditional model multiplies by the number of objects, but ours remains constant or even reduces if some of the class methods are no longer used (as shows the comparison of the second row of Table 1 with the third and fourth rows.)

Table 1 also shows that the OO model of the crossroad problem can be clocked about 27% faster (limited by the farmroad_light$open FU). This clock frequency is also independent of the number of objects and is attributed to the target architecture, while in the traditional methods, this speed even decreases by the number of objects and the resulting complexity. The crossroad OO model owns two objects each having three data members (common state, and elapsed_time data members, plus min_green for the highway_light object, and fixed_green for the farmroad_light object) plus frl_sensor system data sensing the car at the farm-road. Hence, it requires 7 memory locations for objects data which is not required in the traditional model.

Implementing the square lights using traffic_light objects (last row) shows a 1558 gates reduction due to elimination of farmroad_light$open FU. It also requires a simpler, and hence smaller, OMU but a little larger MIU which is due to green-state timekeeping formerly done in the FU and now in the MIU. The two implementations are functionally equivalent but the second requires 33% less area and runs 40% faster. This shows that although our OO synthesis inherently has some area overhead, choice of objects can intensively affect it. Besides, in a traditional modeling paradigm it is natural to need more chip area to control four lights of a square compared to two lights of the crossroad; however, in the OO model we see an inverse relation. This shows that the chip area in our synthesis approach grows with the class hierarchy, not the number of objects. Hence, although in today technology the chip area is not the limiting factor, in large applications where the ratio of number of objects to classes increases the OO area overhead not only can be ignored, but the total area may even be less than traditional approaches.

Above all, it should be mentioned that if the MIU is

```
1 main0() { //streets of the square respectively.
2    turn=0;
3    S.open(); W.open();
4    S.timekeeper(); W.timekeeper();
5 forever do {
6       switch(turn){
7          0: N.open();
8          1: W.open();
9          2: S.open();
10         3: E.open();
11      } //switch
12     turn = (turn++)%4;
13     } //forever
14 } //main0

1 main1() { //Four traffic lights assigned to
2    S.timekeeper(); E.timekeeper();
3    S.open(); E.open();
4    S.close(); E.close();
5 forever do {
6       switch(turn){
7          0: N.open();
8          1: W.open();
9          2: S.open();
10         3: E.open();
11      } //switch
12     turn = (turn++)%4;
13     } //forever
14 } //main1
```
micro-programmed, the two new examples can be realized on the same hardware as the first one by merely updating the micro-memory and the mapping tables of the OMU and MIU. This holds true as long as the class hierarchy is fixed. In fact, for a certain class hierarchy, the synthesized target architecture can be viewed as an implementation platform for any product that can be modeled using object of those classes. This leads to even shorter time-to-market for that family of products.

6 Summary and Conclusion

A novel synthesis approach for implementing functional-level object-oriented models is presented as our main contribution. A new view to mapping of object-oriented modeling concepts (class’s data and methods) to hardware modules is proposed and a novel architecture capable of realizing polymorphism caused by inheritance is introduced. The class methods are viewed as behavioral hardware modules operating on class (object) data stored in a global memory. It is assumed that objects data are laid out in memory in the same way as C++ compilers do, so as to later extend the proposed approach to system-level design of embedded systems in OO methodology. The synthesis flow is independent from both OO specification language and HDL simulation or synthesis language. As a point of emphasis, we motivate the functional-level OO model as the starting point of the flow; i.e., designer should decompose the problem in terms of its natural functional objects rather than structural objects of a certain target architecture. As case studies, two traffic light controllers are modeled in object-oriented methodology to clarify the concept and illustrate the advantages. A graph model is presented to capture objects interaction from which the system schedule is extracted. The design and synthesis flow is illustrated through realizing the first case study in Verilog®, validating it through simulation, and finally synthesizing it on a sample ASIC process. Although the synthesis flow was followed by manual steps, it was shown that it can be automated in an EDA tool. The area, delay, and memory usage figures of synthesizing three OO models are presented and compared to that of a traditional modeling approach.

The experimental results prove that although support for OO features (such as inheritance and polymorphism) introduces some area overhead, the overhead is proportional to the size of the class hierarchy and its methods, and not to the number of objects in the model; therefore, in large applications, where OO shows its best in terms of complexity management and flexibility, the total area can even be less than traditional models. Besides, for a given class hierarchy, synthesized target architecture can be made independent of the application by micro-programming the MIU and storing the mapping tables of the OMU and MIU in a reprogrammable memory. In this way, the same chip can be reused to implement several applications modeled using the same class hierarchy.

The architecture, in its current form, does not support nested (and hence recursive) method calls. Although not required in many designs such as our case studies, this is obviously a disadvantage and must be addressed. A simple solution could be providing proper signals from FUs to the MIU requesting it to activate an (oid, mid) pair. This allows for nested method calls, but will not allow any recursive calls, yet. A good solution requires more investigation.

Parameter passing to class methods is another point for further research. Again, as in our case studies, it may not be always required, but is to be addressed. A simple solution is to define all parameters as data members of the class. But this imposes extra load to the memory and is incompatible with software compilers. This is another point of our further research.

At any point in time, the MIU in the proposed target architecture is aware of the system activities in terms of active FUs, and hence, can turn off inactive FUs to save power. Furthermore, when synthesizing towards capacity-limited platforms, such as FPGAs, the MIU can swap-out inactive FUs and swap-in the required ones to save space. These are other directions for our future research.

There have been several efforts in the past in synthesizing OO descriptions. We believe that viewing “structural elements” as “objects” is the main drawback that has prevented their widespread use. We wish to propose “functional-level OO modeling” as the next step towards higher abstraction levels after “behavioral modeling”; it can lead to a remarkable increase in designer productivity by hiding more implementation details while leveraging currently mature “behavioral synthesis”. “Functional-level OO modeling” models the system in terms of its natural constituting objects. The class methods that operate on those objects are modeled at behavioral level, and hence, can take advantage of any progress achieved in behavioral synthesis. Authors
would like to emphasize that starting the design flow by identifying functional (not structural) objects is a critical point that must be met or OO potential advantages will not be fully obtained.

References: