Journal of Universal Computer Science, vol. 10, no. 9 (2004), 1123-1155
submitted: 7/11/03, accepted: 11/8/04, appeared: 28/9/04 © J.UCS

Object-Oriented Embedded System Development Based on Synthesis and Reuse of OO-ASIPs

Maziar Goudarzi
(Sharif University of Technology, I.R.Iran
and
University of Cambridge, UK
gudarzi@mehr.sharif.edu)

Shaahin Hessabi
(Sharif University of Technology, I.R.Iran
hessabi@sharif.edu)

Alan Mycroft
(University of Cambridge, UK
am@cl.cam.ac.uk)

Abstract: We present an embedded-system design flow, discuss its details, and demonstrate its advantages. We adopt the object-oriented methodology for the system-level model because software dominates hardware in embedded systems and the object-oriented methodology is already established for software design and reuse. As the building-block of system implementation, we synthesise application-specific processors that are reusable, through programming, for several related applications. This addresses the high cost and risk of manufacturing specialised hardware tailored to only a single application. Both the processor and its software are generated from the model of the system by the synthesis and compilation procedures provided. We observe that the key point in object-oriented methodology is the class library, and hence, we implement methods of the class library as the instruction-set of the processor. This allows the processor to be synthesised just once, but, by programming, to be reused several times and specialised to new applications that use the same class library. An important point here is that the processor allows its instructions to be selectively overridden by software routines; this not only allows augmentation of processor capabilities in software, but also enables a structured approach to make software patches to faulty or outdated hardware. A case study illustrates application of the methodology to various applications modelled on top of a common basis class library, and moreover, demonstrates new application-specific opportunities in power management (and area-management for FPGA implementations) resulting from the structure of the processor based on deactivation of unused features.

Key Words: embedded system design, hardware/software synthesis, high-level synthesis, object-oriented system design, application-specific instruction processors
Category: C.5.4, C.3, J.6

1 Introduction

VLSI designers are facing a "quadruple whammy" [August et al. 02]: more transistors per chip, harder to design transistors and interconnects in deep submicron, more heterogeneous elements on the same silicon, and tighter time-to-market deadlines. Hence, chip design, synthesis, and manufacturing is continuously getting harder, more expensive, and more time-consuming. One way to address these is to use application-specific instruction processors (ASIP) instead of application-specific integrated circuits (ASIC). An ASIC is a hardware specialised solely to a single application to provide that application with the best quality factors (e.g. performance, chip area and power consumption); an ASIC may accept some parameters affecting its operation, but is not programmable to adapt to the needs of other applications and hence cannot be used for any purpose other than its target application. On the contrary, an ASIP offers programmability to allow in-field (after the chip is manufactured) customisation to several related applications; this allows the same chip to be reused for different but related applications or for successive generations of the same application. An ASIP is tailored to a set of target applications and hence can be manufactured in larger volumes to reduce the unit price, while allowing programmability through software to shrink time-to-market and ameliorate design risk. However, system implementation using ASIPs risks additional costs of lower performance and higher area and energy consumption when compared to ASICs. ASIP designers strive to mitigate these costs by providing specific hardware features for the application domain being supported. To achieve this goal, the MESCAL project [MESCAL 03] suggests a complete ASIP design methodology as a five-step discipline [Keutzer et al. 02]. This discipline is defined to completely characterise the application domain, fully explore the space of various possible architectures, and provide suitable mechanisms to enable the software efficiently use the hardware. Section 2.2 gives an in-depth comparison of ASIPs vs. ASICs and reviews the MESCAL discipline to build enough background to clarify the position of our work in this research strand.

In our ODYSSEY research project [ODYSSEY 03] on system-level design solutions, we stay in the same roadmap as the MESCAL discipline suggests, but in addition we take advantage of the object-oriented (OO) methodology. Our motive behind advocating OO for embedded-system design is the need for higher abstraction levels, in addition to the higher cost of software design compared to hardware in embedded systems [ITRS 01]. These on one hand emphasise the need for a higher abstraction level while on the other hand imply that a software-suitable methodology is a better choice than a hardware-suitable one for the "system design methodology". In the ODYSSEY project, we aim at enabling the designer to design a high-level OO model of the system-under-design and synthesise it into cooperating hardware and software components; moreover, it is of our primary concern to reuse the synthesised hardware and to program it through software for future extensions of the same application or for implementing new but related ones.

In the rest of this paper, we first present background information on object-oriented design methodology as well as ASIP-based implementation. Then in Section 3, we draw a sketch of the ultimate design flow in the ODYSSEY project and present our proposed flow for single-processor object-oriented design of embedded systems. In an ASIP-based development solution, the three primary issues are how to select the instruction-set, how to synthesise the ASIP with that instruction-set, and how to program the synthesised ASIP. The first two issues are introduced in Section 4 along with details on implementing class polymorphism, and the third one is presented in Section 5. A case study in Section 6 illustrates our view of system-level OO modelling and the reuse methodology; the section also presents and analyses the experimental results of simulation and synthesis of the case studies. Related work is introduced, discussed and compared to ours in Section 7. Finally Section 8 concludes the paper and presents directions for further research.

2 Background

For the last three decades, the ever increasing need for more complex systems and shorter time-to-market has always motivated chip design at higher abstraction levels. Currently, the highest abstraction level that is mature enough to be widely adapted by designers and facilitated by commercial design automation tools is the *behavioural level* where the hardware designer describes the behaviours of the circuit-under-design as sequential algorithms (each very similar to a software program); behavioural synthesis tools then analyse the design and synthesise gates and registers that provide the same functionality.

The need for higher abstraction levels not only is still felt, but also is expected to exist for the foreseeable future [ITRS 01]. *System-level design* is generally believed to be the next step in this move towards higher abstractions. In system-level design, not only the hardware, but also the software of the system, and their interface, is desired to be abstracted. In other words, the system should be designed in an abstract model that is independent of its final hardware or software implementation, then system-level synthesis tools synthesise interacting hardware and software components that provide the same functionality as the system-level model. Naturally, the first question raised is "What is a system model?" and the second one is "How is it synthesised into interacting hardware and software?". In the ODYSSEY project [ODYSSEY 03], we advocate *object-oriented model* as the answer to the first question, and *ASIP synthesis and programming* for the second one. To justify these decisions, this section first reviews object-oriented methodology and compares ASIPs vs. ASICs, and then discusses our decisions.

2.1 Object-Oriented Methodology

In general, two different approaches can be taken to model a given application. The first approach puts the emphasis on the *algorithm* that solves the problem or provides the required service. In other words, the model is a step-by-step path that when followed produces the expected results; such an approach is generally called *algorithm decomposition*. On the other hand, the second approach puts the emphasis on the *data* items and components that constitute the surrounding world of the problem. This approach, also called *data decomposition*, models the application as the interaction of its involved components that together provide the expected results. These data components, or *objects*, of the model often correspond to real-world objects, and hence, provide a more natural and rational way of problem solving. Each object has a set of data fields, or *attributes*, that is (in principle) not accessible to others; a certain set of operations, or *methods* (also called *behaviours*), that each object defines is the sole interface to the other objects and external world. Objects can invoke operations of, or pass messages to, each other. These objects and their message-passing finally provide the required service or result.

Inheritance. To efficiently model objects and their methods, they can be categorised into *classes*; all objects of the same class have the same attributes and methods, but can have their own value for each of the attributes. Some classes of objects may have similarities in attributes and methods. The *generalisation/specialisation* mechanism, also known as *inheritance*, allows to build a hierarchy of classes where each child class inherits all attributes and methods from its parent(s) and adds new ones if required. Therefore, the parent class is a generalisation of all its children and each child is a specialisation of its parent(s). This is a powerful mechanism that facilitates reuse of already defined features as well as management of complexity by incrementally adding new features to existing classes.

Polymorphism. A child class can also redefine each method of its parent to correspond to the special needs of the child. Now, when this same method is invoked on objects of the parent and the child class, the performed operation should differ. *Polymorphism*, or *polymorphic behaviour*, requires that the performed operation depend on the class of the called object; if the object is of the parent class, the parent method should be invoked, and if the object is of the child class, it is the child method that should be run. In general, polymorphic behaviour may not be desired from all methods of a class. The designer can selectively enable polymorphism for each method by annotating the method declaration; such methods, that are to provide polymorphic behaviour, are often called *virtual methods*. Implementing polymorphism requires that when a virtual method is called on an object, the class of the called object is determined, then the implementation of the called virtual method in that class is located, and finally the call is dispatched to that method implementation; thus, polymorphism implementation is also known as *virtual method despatch*.

The above-explained philosophy of design is called object-oriented methodology and the programming style that uses this philosophy is called object-oriented programming. A wide variety of languages (e.g. Java, SmallTalk, C++) provide features required for object-oriented programming; these are object-oriented languages. Each object-oriented language may provide different mechanisms to define the fundamental concepts of classes, objects, inheritance, and polymorphism; however, these concepts are supported by all of them.

What we presented in this section, was a broad view of the object-oriented paradigm for modelling applications. It is out of scope of this paper to talk about how to analyse and design object-oriented applications or how to do it practically. To provide system-level synthesis from OO models, we consider only the fundamental concepts explained above and provide the path to synthesise collaborating software and hardware from models described by those concepts; this gives language-independence to the synthesis methodology. Further reading on object-oriented analysis, design, and programming can be found in books such as [Booch 94, Lee and Tepfenhart 97].

2.2 ASIPs versus ASICs

The technology advancement towards nanometer transistor sizes has made the design of ASICs much harder and much more expensive than ever. This arises from four distinct sources: designing each transistor is harder in deep sub-micron geometries (generally accepted as < 250nm), at the same time exponentially more transistors are to be designed with this higher levels of integration and increased die-sizes, integrating heterogeneous components (digital, analog, and mixed-signal) on the same die is also common now, and in addition, the time-to-market for products is increasingly shrinking. Consequently, the cost for the design of an ASIC in 2010 is estimated to hit \$100M [ITRS 01]. In addition to this high risk and high cost of design, the manufacturing cost is also rising to multi-million dollars for sub-100nm geometries; in the current 180-130nm processes, it is already in the 0.5-1M\$ range and is only to raise with further shrinking geometries.

The aforementioned high cost of ASIC design and manufacturing necessitates a higher production volume to amortise the costs over a larger number and reduce the unit price. Providing “programmability” is one way to achieve higher volumes; it allows the same chip to be used for multiple related applications as well as for various generations of the same application. Moreover, programmability mitigates the aforementioned high risk of ASIC design; writing and debugging an application in software is much cheaper, easier, and faster than designing, debugging, and manufacturing working hardware. Hardware that features programmability is known as ASIP, also referred to as programmable platform. Contemporary evidence, especially in the fields of network and communication processors, confirms that ASIPs are rapidly emerging [Keutzer et al. 02]. However, the flexibility offered by this alternative implementation style to ASIC often comes at a significant cost in performance and power consumption.

What is obviously desired is the flexibility of ASIP with the efficiency of ASIC. To achieve this, a complete design methodology is required that enables the ASIP to provide specific hardware features for the application domain being supported. The research in the MESCAL project [MESCAL 03], aimed at developing a methodology and set of supporting tools for ASIP-based design, suggests such a disciplined methodology that is discussed below.

MESCAL Five-Step ASIP Design Discipline. The MESCAL project breaks the ASIP-based design into two parts. The first part, development of ASIPs, deals with the design of the ASIP hardware so that better service is provided for the supported applications while providing software features that allow efficient programming. The second part, deployment of ASIPs, talks about how to effectively use and program ASIPs to implement applications. The latter is part of their current and future research but for the former, they have outlined the following five-step methodology [Keutzer et al. 02]:

1. *Disciplined Benchmarking.* An ASIP is, by definition, tailored to an application domain. This requires a set of benchmarks for analysis to identify required features of that domain. Traditionally, benchmarks have consisted of application kernels, and the only quantitative measure used has been the number of execution cycles. MESCAL requires that benchmarks must consist of a functional specification to describe what it should do, a requirements specification to specify what requirements the application must meet, an environmental specification to ensure a complete and well-defined set of stimuli, and a set of measurable performance metrics such as power consumption and memory bandwidth in addition to execution time; moreover, the benchmarks must be at the application-level rather than the kernel-level as kernel-level benchmarks can potentially hide performance bottlenecks. One such disciplined benchmarking is presented in the context of network processors in [Tsai et al. 02].

2. *Defining the Architectural Space.* A wide variety of architectures are possible to develop an ASIP. However, not all of them may be suitable for a given domain of applications. A careful definition of the suitable space of architectures is essential for the subsequent design space exploration.

3. *Efficiently Describing the Detailed Design Space to be Explored.* After the benchmarks and architectures are defined, all benchmarks should be mapped to each architecture and the design metrics should be measured to systematically explore the design space and find the best choice. Obviously, this mapping cannot be done from scratch for each architecture; retargettable software tools (e.g. compilers, instruction-set simulators, and power simulators) are required to easily change target to another architecture. In addition, it is essential that all these software tools stay consistent to each other and to the target architecture in consideration. One way to achieve this, is to define an efficient architecture description language from which all software tools are generated. MESCAL introduces the MESCAL Architecture Description (MAD) language along with metatools to generate compiler, simulators, and the like tailored to the architecture being considered.

4. *Exploring the Design Space.* The design space is explored in an iterative process of compiling the benchmarks on the design point, evaluating the result on the metrics using the simulator and accordingly enhancing the ASIP design. The retargettability of the tools is the key to efficiency of this process for a large number of design points.

5. *Exporting the Programming Environment for the Final Architecture.* The application-specific features of an ASIP may be completely useless if it cannot be programmed efficiently. Efficient programming requires a high level language for today's levels of software complexity. Furthermore, language features are required that expose those details of the architecture that account for most of its performance. A long-known example is the C language; C exported 20% of the architectural features of 1970s processors (such as pointers and registers) that resulted in 80% of their performance [Keutzer et al. 02]. The modern equivalent of such a programmers' model is needed that exports critical features of the ASIP to the programmer for easy and efficient exploitation.

2.3 Justification of the Methodology Decisions

As discussed in the first part of this section, elevating the design abstraction level is inevitable in order to cope with the design complexity as well as the market demands. "System-level design" is the next milestone in this path and a design methodology is required that supports modelling at this level and synthesising lower-level components from that model. Such a methodology can start from a hardware-resembling model (e.g. SystemC [SystemC 04] and SystemVerilog [SystemVerilog 04]), or a software-resembling model (e.g. Java [OASE 03]), or a mathematical model (e.g. co-design finite-state machine [Sgroi et al. 00]). Any of these choices has its own advantages and disadvantages, and hence, each can make sense depending on the point of emphasis. In safety critical systems, a strict mathematical model can be of more interest since it enables strict characterisation and assessment of safety and fault-tolerance factors. In computation-intensive applications, a hardware-centred model can be a better fit that first provides the necessary performance but can also give enough flexibility to allow slight modifications or upgrade. In rapidly-changing market sectors, where the customer demands constantly change and a few additional features can result in a big increase in market share (e.g. mobile phones, personal digital assistants, and other general consumer electronics), a software-centred model is more appealing due to the ease of modification and upgrade it offers. In such general embedded systems, which we envisage as our target applications, short time-to-market is required and a tight market-window applies. Moreover, the International Technology Roadmap for Semiconductors reports that software now routinely accounts for 80% of embedded system development cost [ITRS 01]. This necessitates paying special attention to the software component of the system. Therefore, we advocate a software-centred model as the initial system-model. Subsequently, within this category, we have chosen the object-oriented methodology. Object-oriented technology has gained such high reputation in the software community that it is believed to be “the most important evolution (revolution) of the 1990s” [Lee and Tepfenhart 97]. Some merits of OO can be listed as capability to share and reuse code, capability to localise and minimise the effects of modifications, capability to manage complexity, and inherent support for concurrency [Cooling 03]. Specifically, support for reuse is a primary issue in OO and is inherently provided by the class and class hierarchy concepts. We take advantage of this property in Section 4 to form a correspondence between a class library (the enabling concept for high-level reuse) and an ASIP (the low-level unit of reuse).

The other important decision to make is the implementation philosophy. In Section 2.2, we presented several facts that have led to the increase in the significance of programmability in chips. For the same reason, we have chosen an ASIP style of implementation. Satisfactorily, this implementation methodology is in harmony with our modelling methodology: both OO and ASIP put emphasis on reuse and extensibility. Section 4 shows how we have put enough hooks in the ASIP hardware to enable capturing the extensions of the model as software of the ASIP. We call such an ASIP, that also provides special support for object-orientation, an OO-ASIP.

3 The ODYSSEY Embedded System Design Flow

In the ODYSSEY project [ODYSSEY 03] we advocate system-level design of embedded systems in the OO methodology. The final system design flow we envisage starts from a high-level implementation-neutral modelling environment, e.g. using UML, that specifically allows concurrency in the model. The concurrency can be both inter-object (multiple objects exist and interact in the system) and intra-object (concurrent method invocations over a single object). ODYSSEY ultimately aims at mapping such a high-level model into a network of OO-ASIPs in an NoC (Network-on-Chip) platform. The motivation behind targeting NoC is that firstly it addresses design issues in deep sub-micron technologies [Benini and DeMicheli 02] and secondly the NoC paradigm matches the OO one: the OO paradigm clearly separates functionality (method implementation) from communication (method invocation); similarly, the NoC paradigm separates computation (processing in network nodes) from communication (packet routing).

In the ideal design environment we currently envisage, first the system-under-design will be modelled in object-oriented methodology where objects can exist concurrently and exchange messages to ask for services from one another, to exchange information, and/or to synchronise their operations. Then C++ code is generated from this concurrent system model, objects are statically partitioned into sets each assigned to an OO-ASIP, and finally the methods of the objects assigned to each OO-ASIP are partitioned into either hardware or software.

In this starting-point work, for simplicity we mainly address system development on a single OO-ASIP, which obviously implies sequentialisation in method invocations due to having a single instruction stream; however, fine-grained concurrency is still provided inside each method implementation (see Section 4).

Single-Processor Embedded System Design Flow. Our proposed design flow is shown in Figure 1. It is divided into two related sub-flows. First, illustrated at the left-hand side of the figure, an OO-ASIP is designed and tailored to the application domain in question. This results in a database consisting of pairs of an OO-ASIP and its corresponding class library; each pair is designed for a certain domain of applications. Later when developing an application, an appropriate OO-ASIP is selected from the database and deployed in the system, shown in the right-hand side of Figure 1. As any other ASIP, an OO-ASIP may cause some performance penalty; however, since OO-ASIP scales well when the applications (and the corresponding class library) grows, such a design flow is very promising.

The “OO-ASIP Design Flow” starts with designing a suitable class library for the application domain. After a disciplined benchmarking and analysis phase (Figure 1, the box with number 1 in the upper left corner), such as the MESCAL proposal, essential abstract data types of the application domain are identified and modelled in a class library called “hardware class library”; critical operations in the application domain comprise the to-be-in-hardware methods of the class library. Then in box 2 of Figure 1, the “hardware class library” is synthesised to an OO-ASIP as described in [Goudarzi et al. 03] and summarised in Section 4.

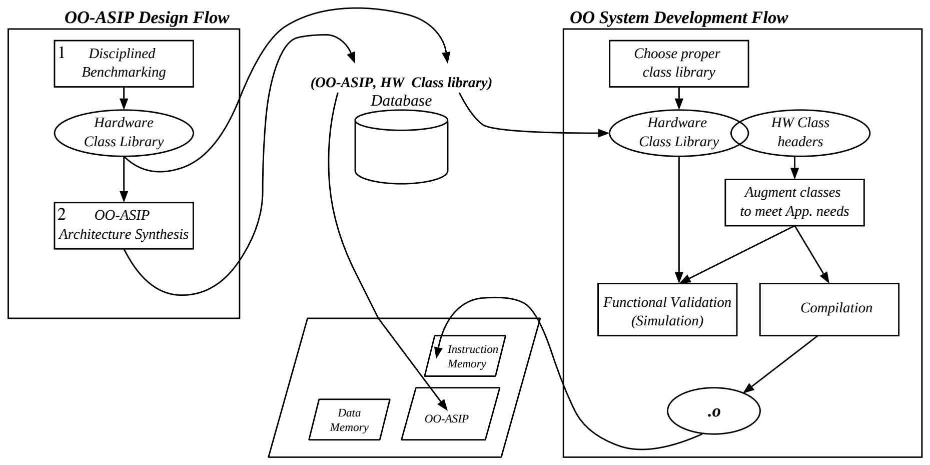

OO-ASIP Design Flow
1 Disciplined Benchmarking
Hardware Class Library
2 OO-ASIP Architecture Synthesis
(OO-ASIP, HW Class library)
Database
OO System Development Flow
Choose proper class library
Hardware Class Library
HW Class headers
Augment classes to meet App. needs
Functional Validation (Simulation)
Compilation
.o
Instruction Memory
Data Memory
OO-ASIP

Figure 1: Object-oriented embedded system design flow using a single OO-ASIP.

The "hardware class library" and its corresponding OO-ASIP are saved in a central database for later adoption and use in the "OO System Development Flow", the right-hand side of Figure 1. This flow starts by choosing a suitable "hardware class library" from the database. The adopted class library may be augmented if necessary to cover application requirements. The extending classes are called "software class library" and the entire resulting classes are named "system class library". We naturally call the methods that have been implemented in hardware as "hardware methods" and similarly those implemented in software as "software methods".

The embedded application is modelled using the "system class library". The model is functionally verified in the designer's favourite environment using the (possibly pre-compiled) "hardware class library" which is functionally equivalent to the OO-ASIP. Finally, compilation (of the software-method parts of the class library) results in an object file to be stored in instruction-memory on the final chip/board along with the synthesised OO-ASIP retrieved from the central database.

4 The Object-Oriented ASIP

Reusability for a *set* of applications is the key property that an ASIP needs to provide, otherwise its overhead compared to ASICs cannot be justified. This firstly requires the ASIP to provide support for the operations that are common in most of the applications in the target set, and secondly facilities must be provided to allow applying post-manufacturing extensions to the ASIP so that required, but unforeseen, features can be provided for the applications that were unavailable at the ASIP development time. Furthermore, to pay primary attention to reuse, we have adopted OO, which is an inherently designed-for-reuse model, as the initial system model in the ODYSSEY project (see Section 2.3). We observe that in an OO model, the methods of the class library constitute all the (macro)operations that can be performed in the model, and moreover, this continues to hold for any application that uses that class library. This can address the first issue above, i.e. identifying common and core operations of the target domain; such operations are the methods of the class library that represents the target application domain. Therefore, we propose to use these methods as the instruction-set of the ASIP. For the second issue, i.e. applying extensions to support unforeseen features, we observe that in the OO methodology, the class library can be extended by new classes that add new attributes and methods to the existing ones. These new methods were not available at ASIP development time, and hence, are not covered in its hardware; however, if the ASIP enables such extensions to run as new parts of the ASIP software, the second issue is also addressed. In this section, we introduce such an ASIP and show how it provides the above capabilities.

In a previous publication [Goudarzi et al. 03] we identified method invocation (among objects in an object-oriented specification) with instruction execution in a processor. In other words, we viewed a method invocation on an object as an instruction to be executed specifying the object and additional argument(s) as instruction operands. This view results in identifying selected methods of the class library as the instruction-set of a corresponding processor. Such a processor will be able to execute any program comprised of objects of that class library, i.e. all applications with that class library. We call this processor an object-oriented application-specific instruction processor, or an OO-ASIP for short. This view results in associating the OO-ASIP with the class library.

A currently implemented internal architecture for the OO-ASIP is shown in Figure 2. Each "hardware method" is implemented as a Functional Unit (FU) shown in the middle of the figure. The OO-ASIP in Figure 2 corresponds to a class hierarchy of two classes: A and B; class A has defined a single $f()$ method while class B, derived from A, overrides the $f()$ method and introduces $g()$ method. The below C++-like code fragment illustrates this simple class library:

```
class A { virtual int f() {...} ...; };
class B extends A {
   virtual int f() {...}
   virtual int g() {...}
   ...;
};
```

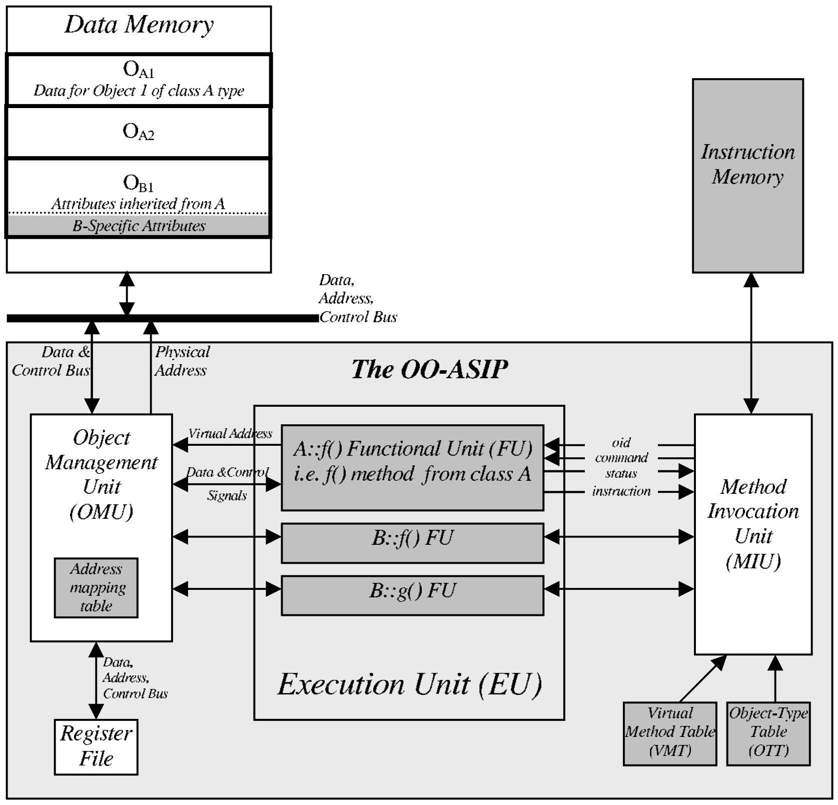


Figure 2: OO-ASIP internal architecture for a sample class library. Here, class B is derived from class A while redefining its $f()$ method and introducing a $g()$ method.

Thus, three distinct methods ($A :: f()$, $B :: f()$, and $B :: g()$) exist in this class library and correspondingly, three FUs are implemented in the OO-ASIP as its Execution Unit (EU). The two other key components of the architecture are the Method Invocation Unit (MIU) and the Object Management Unit (OMU) described below. All program instructions (method invocation commands) are stored in an instruction memory shown at the upper right of Figure 2. During OO-ASIP operation, instructions are read from the instruction memory by the MIU; each instruction designates a method (opcode) and one or more arguments (addressed by instruction operands); as conventional in OO style, the first argument indicates the object to be manipulated. The MIU determines (see below) and activates the corresponding FU implementing the method in question. The MIU can also dispatch such instructions to software routines depending on the class of the called object; this class membership of the object is determined at run-time to enable polymorphism (see "Implementing Polymorphism" part below).

All objects' data fields are stored in the data memory at the upper left of Figure 2. Objects of the same class (e.g. O_{A1} and O_{A2} that are respectively objects 1 and 2 of class A) have the same size and layout for their attributes. Objects of a derived class (e.g. O_{B1} as object 1 of class B derived from class A) keep the same layout for their inherited attributes, but append it with their newly added fields (the grey part labeled "B-Specific Attributes" in the O_{B1} box).

In an OO model, a single method may be called on different objects, and hence, a mechanism is required to enable the method implementation to access all objects that may be called. Correspondingly, each FU (the method implementation in the architecture of Figure 2) must be able to operate on all appropriate objects; this is provided by the OMU that regulates all accesses to object data fields. To access fields of the called object, the FU communicates with the OMU which makes the memory or register transaction on behalf of the FU. The OMU also implements a data cache to accelerate data access. A "hardware method" may need to call other methods to accomplish its task. In such a case, the active "hardware method" issues an *internal* instruction (through the `instruction` signal between the FU and the MIU in Figure 2) and the MIU dispatches it in the same way as normal instructions read from the instruction memory.

Our key contributions in that previous publication [Goudarzi et al. 03] are to define the instruction-set of an ASIP by *selected* methods of the "hardware class library", to propose an architecture to implement the OO-ASIP, and to realise polymorphism in hardware, supporting (virtual) method dispatch even to software. The synthesis process can in fact be expanded to steps 2 to 4 of the MESCAL discipline. The proposed internal architecture (presented in Figure 2) is only a prototype to present the concepts and show their use in practice; the fine details of this architecture are not our primary focus in this paper and hence are not discussed anymore. However, the implementation of polymorphism is presented below due to its importance. In addition to being a very important feature of the OO methodology, we show in Section 5 that polymorphism can be used to apply software patches to hardware after manufacturing.

Implementing Polymorphism. We define the following notation: each object in the system is designated by an "object identifier" or *oid*, each class by a "class identifier" or *cid*, each class method by a "method identifier" or *mid* (shared by all redefinitions of that method in child classes), and each FU by a "Functional Unit identifier" or *FUid*.

Polymorphism implies dynamic type checking of the called object to find out the method that corresponds to the instruction being executed. This is done by the MIU through the *Object-Type Table (OTT)* and *Virtual-Method Table*

(VMT) in Figure 2. The OTT shows current class membership of all objects available in the system, i.e. an $oid \rightarrow cid$ mapping. In software realisations, this information is normally stored as a (hidden) tag in the object attribute storage. We hold this inside the OO-ASIP for higher performance, but this is not particularly important here.

The VMT is a matrix containing an entry for each class of the system and for each virtual method, designating its corresponding FU. Hence, it is a mapping function of type $(cid, mid) \rightarrow (FUid)$. Naturally, the row of a derived class is identical to its immediate parent except for its overridden and newly introduced methods.

OO-ASIP instructions designate the *mid* (the opcode, i.e. the called method) and the *oid* (first operand, i.e. the called object). After reading an instruction, the MIU consults the OTT to find out the *cid* of the called object, and then maps the (cid, mid) pair to an $FUid$ through the VMT. The resulting $FUid$ is not necessarily a hardware unit but may point to the starting address of a software routine according to a tag bit; in either case, it is the appropriate method of the called object and is invoked accordingly. This entire process effectively realises polymorphism.

It is worth considering how the virtual method dispatch mechanism works as the source and destination of the call vary between hardware and software implementation. Non-virtual calls are just a special case where we know the implementation format of the callee. It was explained above that the despatch mechanism works irrespective of the implementation format of the caller and callee. However, passing parameters to the method implementation can be tricky depending on the caller and callee implementation format. Two alternative mechanisms for this parameter passing are: pushing parameters on a stack, or writing them to argument registers. The stack-based approach provides independence from caller and callee implementations, while the register-based approach gives higher performance. These are discussed below.

First assume all actual arguments are put on a stack before each method call. This provides the same mechanism for parameter passing irrespective of the caller and callee being in hardware or software. For example, assume a software method `f()` invokes a call `obj.g()` while `obj` is an object of class B; `g`'s arguments are pushed on the stack. The `g` component of the VMT for B is then extracted. This is then (according to a tag bit) either interpreted as a software method (in which case the call effects a traditional branch-and-link instruction) or as a hardware method in which it effects an OO-ASIP instruction B`::g`. The same would hold for a hardware caller.

Now assume that registers are used for parameter passing instead of a stack to gain higher performance. For a software caller, the scenario is simple and the same as above, but the parameters are placed in assumed registers `a1`, $\ldots$, an. For a hardware caller, however, two cases would arise depending on the implementation format of the call destination: a hardware destination just corresponds to the activation of the appropriate FU as above (and the movement of arguments to the callee's input register); a software destination is more tricky as the call arguments (held internally in the caller's FU) must be routed to the input registers (a1, ..., an as above) for `g()`.

It is worthy of note that the register-based calling scheme above means that each method declaration (and all its overriding instances—i.e. the same *mid*) uses a common set of argument registers, but different method declarations (i.e. separate *mid*s) use separate argument registers. This simplifies the software/hardware interface for presentational purposes; in practice we would seek to use a single set of argument registers (cf. procedure call on MIPS or ARM processors), but doing so is more complicated and puts additional requirements for a software callee to save argument registers before using a hardware instruction which could use a software method. This is not a concern when using a stack for parameter passing.

5 Programming the OO-ASIP

As described in previous section, we synthesise a class library as an OO-ASIP. Applications modelled with that hardware class library without augmenting it run directly on the OO-ASIP, e.g.

```
A x; // Class A is part of the "hardware class library"
main() { x.init(); x.go(); }
```

corresponds to two instructions of the OO-ASIP since A`::init()` and A`::go()` are hardware methods, and hence, are instructions of the OO-ASIP. However, evolution of the products and applications implies extensions to the class libraries (e.g. to provide more complicated functionality using the existing base classes, or to override a hardware-implemented method with an upgraded software implementation). Figure 3 gives a diagrammatic view of the system class library where its hardware portion (in grey box) corresponds to the OO-ASIP, and the extending classes (in white box) correspond to the software for the OO-ASIP. Software methods either call each other (this just corresponds to OO-ASIP software function call), or call hardware methods (this just corresponds to an OO-ASIP single instruction). Therefore, software methods can be translated into a sequence of instructions representing hardware and software method calls, and thus are the software for the OO-ASIP, for example stored in flash ROM. Traditional processors can be viewed as special cases of the general OO-ASIP, performing operations on simple objects such as `int, float,` etc.; hence, running general software is possible by including FUs corresponding to traditional

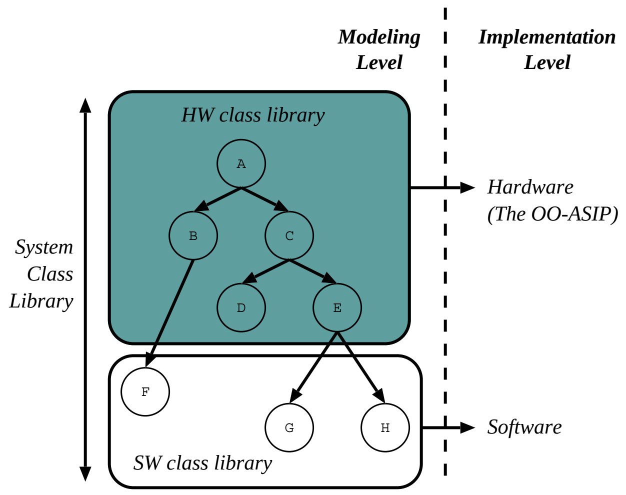


Figure 3: Diagrammatic view of the correspondence between the OO-ASIP and the hardware class library, and also between the software class library and the OO-ASIP software.

ALU-like operations, or alternatively by supplying a general-purpose processor core along with the OO-ASIP.

Note also how OO model of evolution and the correspondence of the OO-ASIP to the hardware class library provides *hardware-accelerator reuse*. An OO model evolves *incrementally* over an existing class library by adding missing functionalities and reusing the *fixed existing ones*; the OO-ASIP, which can be viewed as a hardware-accelerator for all applications modelled with that hardware class library, corresponds to that fixed existing part and hence directly addresses reusability in future application extensions.

To illustrate embedded system development and evolution using the OO-ASIP, consider the example of a security system controlling and monitoring authorised access of people to rooms. *People* and *rooms* are the main objects in the system. Access rights, authentication algorithm, and recorded information may differ for each person, and similarly for each room. Therefore, one would design two class hierarchies for *people* and *rooms* and synthesise an OO-ASIP for them. To implement a specific instance of security system, the designer instantiates appropriate people and rooms and properly models their interaction by appropriate method calls over objects.

As long as no new type of *people* or *room* is introduced to the system, i.e. the hardware class library suffices, the same OO-ASIP can be reused for new security systems, even if the number of objects or the sequence of method calls change (e.g. recording of the access trials is now to be done for every attempt instead of merely after successful authentication); such changes correspond to software modifications of the OO-ASIP.

However, if for example a new type of *room* is added to the system, the class library of *rooms* needs to be augmented. This is done by adding software classes. The new *room* either still uses OO-ASIP-provided methods but in a different way (e.g. uses triple-DES instead of simple DES[1] for authentication, allowing reuse of current hardware methods), or radically differs in behaviour. In this latter case, the worst case, the new behaviour is implemented in software, using current class methods wherever possible. Performance degradation for objects of these new classes is the price one pays for the unforeseen support in this worst case of class augmentation. However, this is a graceful degradation occurring only to the objects of the new type. Moreover, note that many time-consuming operations can still be dispatched to hardware methods even if an application uses only objects of software classes; e.g. in the domain of digital signal processing applications a MAC (Multiply and ACcumulate) method in the hardware class library enables all future software methods to use it.

The design of *people* and *rooms* class hierarchies is the starting point for the development of an OO-ASIP tailored to the domain of security systems, and hence, it plays a significant role in practicality of the final ASIP. We wish to leverage the MESCAL efforts in disciplined benchmarking here for designing the hardware class library, and hence, we do not concern ourselves with the process of developing the hardware class library anymore in this paper. The MESCAL disciplined benchmarking is meant to carefully analyse the target application domain; therefore, part of its results can be captured as our hardware class library.

Another programming feature of the OO-ASIP compared to traditional processors is that it enables the programmer to selectively override hardware units by software routines; e.g. the following code extends the example shown at the beginning of this section by deriving a software class B from the hardware class A and overriding its $go()$ method:

```
class B extends A {
   void go() {...}  // overrides A::go() method
};
B x;
main() { x.init(); x.go();}
```

Although the code in the $main()$ function is the same as before, the $x.go()$ method call is now dispatched to $B :: go()$ instead of $A :: go()$. This shows how

[1] DES (Data Encription Standard) is a cryptography method that operates on fixed-sized blocks and generates blocks with the same size. Triple-DES uses three subsequent DES operations for higher security.

software classes can selectively override hardware methods. This feature may be required in several circumstances where, for example, the hardware unit has a physical, manufacturing, or design fault or has gone out of date and replacing it is too expensive or even impossible (e.g. when the hardware is installed in an out-of-reach satellite or spacecraft). The traditional approach to this problem is to rewrite, in an ad hoc manner, the entire program in order to catch all invocations of the faulty hardware and to replace them by correct software routines. Note how the use of dynamic method dispatch provides a structured approach to incorporating enough 'hooks' in hardware to allow software patches to out-of-date features; this is to be contrasted with the current ad hoc approach explained above. These 'patches' can override an outdated or faulty-designed (or manufactured) hardware FU with correct software implementation (i.e. *post-manufacturing design-error correction*) in return for a performance penalty that conveniently only affects objects that could exercise the faulty FU.

5.1 Functional Verification of the Application

The correspondence between an OO-ASIP and its "hardware class library" allows to use the class library as the software counterpart of the OO-ASIP when designing new applications to run on the OO-ASIP. The designer selects an appropriate class library, models the application in his favourite design environment, and functionally validates it by simulation on his favourite OS-processor platform.

Imagine an application programmer building a new instance of security systems following the above example. Access to *people* and *rooms* class hierarchies is enough for him to design the application and functionally validate it. Regardless of the design environment, the class library/OO-ASIP correspondence means the application has been functionally validated on the OO-ASIP.

5.2 Compiling the Application

5.2.1 Embedded OO Programs—Numbering Concepts

In object oriented systems, there are two separate concepts concerning associating distinct identifiers (e.g. small integers) to methods. One is to allocate *per-definition*, so that `A::f()` and `B::f()` are given separate identifiers. We refer to these as functional unit identifiers (*FUid*, introduced in Section 4). The other is to allocate *per-declaration* where overriding virtual methods are given the same identifier as the overridden method. These latter will be referred to as method identifiers (*mid*). OO-ASIP method invocation instructions will use the *mid* form, which can then be dynamically dispatched to the *FUid* form. Note that the *mid* form is determined for a "hardware class library" irrespective of how it is extended, a fact which we exploit in the following section in building an OO-ASIP.

Implementing polymorphism requires identical encodings (i.e. *mid*) for a virtual method and all its overriding definitions throughout the library. Numbering of methods starts from the root node of the library. Whenever encountering a new method (not a redefinition) within the library a new number is assigned to it which is also inherited to all its subsequent redefinitions. These *mid* encodings along with operand encodings in the instruction, number and type of registers, and support for dynamic object allocation/deallocation and the like, are characteristics of the OO-ASIP and are saved in the *(OO-ASIP, class library)* database of Figure 1. The compiler uses these for code generation.

We have not yet implemented a retargettable compiler for the OO-ASIPs and currently assemble the programs manually. However, automatic compiler generation from the *(OO-ASIP, class library)* database entry is possible and will be addressed in the ODYSSEY project—the task consists of straightforward modifications to the compiler algorithm which replaces calls in intrinsic functions to hardware instructions; in the same way that a compiler maps a language call to (say) `sqrt` function by loading arguments into registers and then using the `SQRT` opcode rather than a `CALL` instruction, we need to map language method invocations of "hardware class methods" into their corresponding opcodes.

5.2.2 Compiling Software Methods

The "software methods" may either override existing "hardware methods" or introduce brand new methods not foreseen when the OO-ASIP was designed. In the former case, the encoding of the method is the same as its overridden hardware one, and hence can be encoded as the corresponding old instruction; updating the VMT is enough to enable the MIU to dispatch method calls also to this new implementation. However in the latter case, as no old instruction corresponds to this new method, the instruction-set of the OO-ASIP needs to be augmented if the MIU is still to be the dispatcher. This ISA-augmentation is possible by reserving a certain amount of encodings in the opcode field of instructions at the OO-ASIP synthesis time; however, this is just a partial remedy. To address this issue when the ISA can no longer be augmented, we switch back to the software implementation of polymorphism, where an indirect `CALL` instruction through a virtual method table branches to the appropriate routine. This allows the "system class library" to grow indefinitely.

6 Traffic Light Controllers—A Case Study

The case study is summarised as follows: a controller is to be designed for the crossroads 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 `min_green` time, it is closed and the farm road temporarily opened for a fixed `fixed_green` time after which the highway is reopened. It is noteworthy that this example is a metaphor for a reasonable design and is only to illustrate the design flow and concepts.

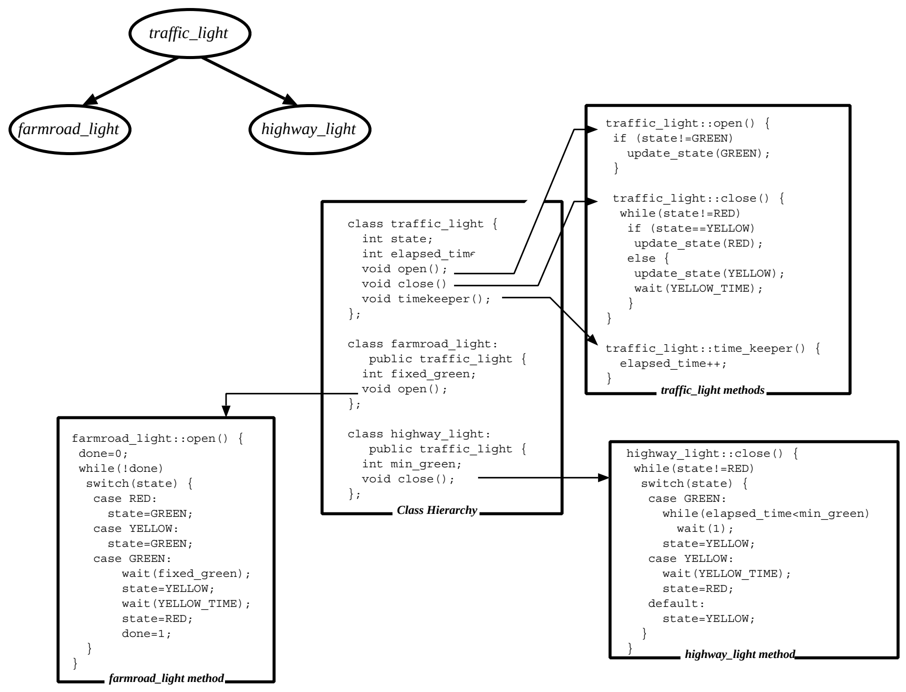
6.1 Experiment Setup

We intend to use a single linguistic framework (C++/SystemC) for both software and hardware components throughout the system design flow, thereby simplifying late-in-the-design-process allocation of methods into hardware or software. However when the implementation of this case study started, we had no access to SystemC synthesis tools. Therefore, Verilog was used to synthesise the OO-ASIP and produce the experimental results[2]. We used ModelSim© and LeonardoSpectrum© to respectively simulate and synthesise the models. Power estimations are reported by PowerCompiler© tool at the register-transfer level based on activity annotations gathered during a sample simulation run.

6.2 Synthesising a *Traffic-Light Controller* OO-ASIP

For this simple example, we propose the class library at the top-left corner of Figure 4 rooted at `traffic_light` class with two child classes `farmroad_light` and `highway_light`. The `traffic_light` class represents a general traffic light that can be green or red for any arbitrary time span, while the `farmroad_light` can be green only for the given fixed time of the farm track. The `highway_light` class is another special kind of `traffic_light` that cannot be made red unless the given minimum green-time of the highway is already elapsed. The central panel in Figure 4, with "Class Hierarchy" caption, shows the declarations of these classes in C++. The `traffic_light` class defines a `state` data member (showing whether the traffic light is in red, green, or yellow state) and an `elapsed_time` data member (showing how long the current state has been active), and three methods open(), close(), and timekeeper(), which respectively turn the light to green, turn it to red, and keep track of time units. These methods are defined in C++ notation in the "traffic_light methods" panel in Figure 4. An auxiliary method, `update_state`, is defined but not shown in the figure; it updates the object `state` field and resets the `elapsed_time`. The `traffic_light::open()` method turns the light to green irrespective of its current state, but the `close()` method ensures that if the light is in green state, it is first turned to yellow and

[2] This interestingly confirms that support for object-orientation is possible even with non-OO HDLs, in the same way as general processors run OO software although they are not OO themselves.



```
class traffic_light {
  int state;
  int elapsed_time
  void open();
  void close()
  void timekeeper();
};

class farmroad_light:
   public traffic_light {
  int fixed_green;
  void open();
};

class highway_light:
   public traffic_light {
  int min_green;
  void close();
};
```

```
traffic_light::open() {
 if (state!=GREEN)
   update_state(GREEN);
 }

 traffic_light::close() {
  while(state!=RED)
   if (state==YELLOW)
    update_state(RED);
   else {
    update_state(YELLOW);
    wait(YELLOW_TIME);
   }
}

traffic_light::time_keeper() {
  elapsed_time++;
}
```

```
farmroad_light::open() {
 done=0;
 while(!done)
  switch(state) {
   case RED:
     state=GREEN;
   case YELLOW:
     state=GREEN;
   case GREEN:
       wait(fixed_green);
       state=YELLOW;
       wait(YELLOW_TIME);
       state=RED;
       done=1;
  }
}
```

```
highway_light::close() {
 while(state!=RED)
  switch(state) {
   case GREEN:
     while(elapsed_time<min_green)
       wait(1);
     state=YELLOW;
   case YELLOW:
     wait(YELLOW_TIME);
     state=RED;
   default:
     state=YELLOW;
  }
}
```

Figure 4: The class library (top-left) along with class definition details (panels) for the traffic-light controller problem.

kept for a certain `YELLOW_TIME` time units before being changed to red. The `traffic_light::timekeeper()` method simply increments the `elapsed_time`.

There are two variants of this class: farmroad and highway lights. Hence, two classes are derived: `farmroad_light` and `highway_light`. The former adds a new data member named `fixed_green` and specialises the open() method, shown in the "farmroad_light method" panel in Figure 4. Following the aforementioned functionality of the farmroad light, this method first turns the light to green and keeps this for the given time of its `fixed_green` attribute, then closes the farmroad again by turning the light subsequently to yellow and red. The `highway_light` class adds the new `min_green` data member and overrides the close() method, shown in the "highway_light method" panel. This method ensures that the light has been green for at least the given `min_green` time before closing the highway by turning the light to red. Details of how to synthesise the above class library into an OO-ASIP are presented in the Appendix A.

The entire model was first validated by simulation and then synthesised. The area/delay results of synthesis over a sample 0.5 μm process technology are shown in Table 1. An analysis of the synthesis figures in Table 1 along with the software

Area (equivalent gates)								*Frequency (MHz)*
Total	*MIU*	*OMU*	*Functional Units*					
			traffic_light			farmroad_light	highway_light	
			open	close	timekeeper	open	close	
7115	460	1845	429	887	564	1558	1372	98.6

Table 1: Results of synthesis of an OO-ASIP for the traffic-light controller problem over a sample 0.5 μm process technology.

```
1  #include "traffic_lights.h"

2  farmroad_light frl(1);
3  highway_light hwl(5);

4  main() {
5   frl.close();
6   hwl.open();
7   forever do {
8    if(frl_sensor) {
9      hwl.close();
10     frl.open();
11     frl.close(); // unnecessary
12     hwl.open();
13   } //if
14  } //forever
15 }
```

main program

```
1  #include "traffic_lights.h"

2 main() {
3  traffic_light *p;
4  for all objects obj do {
5   p = & obj;
6   p->timekeeper();
7  }
8 }
```

timekeeper interrupt routine

Figure 5: Software for the highway-farmroad traffic-lights problem.

running on the OO-ASIP (shown in Figure 5) is presented in Section 6.5.

6.3 Developing *Traffic-Light Controller* Applications

The C++ source code for the highway-farmroad problem is presented in Figure 5. The "main program" panel shows two static object instantiations, `frl` and `hwl` for the two traffic lights; line 2 states that the `fixed_green` field of the `frl` object is initialised to 1, while line 3 initialises the `min_green` field of the `hwl` object to 5. The `main()` function first initialises the system so that the farmroad is closed (line 5) and the highway is opened (line 6) and then defines the normal operation (the `forever` loop in lines 7 to 14). A sensor is installed in the farmroad to detect whether a car is waiting there to cross the highway; `frl_sensor` represents the status of this sensor. If this sensor shows that a car is waiting there (line 8), first the highway is closed (line 9) and then the farmroad is opened (line 10); then, the reverse is done (lines 11 and 12) to return to the normal state. Recalling definitions of the class methods in Figure 4 makes it clear that line 9 ensures that the highway is kept at least 5 time units open before being closed again; also note that `frl.close()` (line 11) need not be called since `frl.open()` is defined to close the farmroad after the given fixed time of 1 time unit. The `frl_sensor`

	Memory Usage		*Power Consumption*[3] *(nW)*	
	instruction *(words)*	*data* *(bytes)*	*without power-down*	*with power-down*
highway-farmroad crossing	8	2×2	171.0	N.A.
Crossing with four lights of type `farmroad_light`	17	4×2	195.2	165.3
Crossing with four lights of type `highway_light`	17	4×2	187.3	148.7

Table 2: Experimental results of software development for the *traffic-light controller* OO-ASIP.

(line 8) is an input to the system and can be a memory-mapped IO port in a physical implementation; similarly, the status of the lights are outputs of the system. Line 8 is currently executed by the MIU, however, `frl_sensor` can be considered an object of class `int`, and hence in a purely-OO system, would be executed by FUs of the default system classes (`int, float`, etc.).

The right-hand side panel in Figure 5 shows the interrupt routine that whenever invoked by the system timer interrupt updates the `elapsed_time` field (line 6) of all `traffic_light` objects (line 4) in the system. Note how lines 4 to 7 use identical commands to call `timekeeper()` method of different objects; the dynamic dispatch mechanism of the OO-ASIP ensures that the method corresponding to the class of the called object is invoked. In this simple example, all three classes use the same `timekeeper()` method, and hence, all iterations of line 6 are dispatched to the `traffic_light::timekeeper()` FU. However, one may derive a new class from `traffic_light` that operates irrespective of the `elapsed_time` (for example controlled manually by a policeman), and hence would override the `timekeeper()` method to do nothing; still the same code in the "timekeeper interrupt routine" panel of Figure 5 works fine and invokes this new method for objects of this new class.

The prototype system implementing the application presented in Figure 5 was machine-coded manually. The results are presented in the first data row of Table 2. The resulting Verilog model was simulated with a set of sample inputs to gather activity information for signals. The model and this activity information were then given to the power estimation tool that reported the number given in the third data column in Table 2. The fourth data column is unavailable in this example since the power management approaches presented in next subsection are not applicable.

[3] As estimated by PowerCompiler tool, over a 1 μm process with 5 V operating voltage.

6.4 Reusing the OO-ASIP

Now, consider a different example of traffic lights at the junction of four roads at a square. Suppose that all roads are of fixed priority and hence fixed green-time. The basic types in the system are the same as highway-farmroad traffic-light controller problem. This encourages the designer to reuse the OO-ASIP developed for that problem. Moreover, any of the three flavours of the light available in the "hardware class library" can be used to model this application, offering a level of freedom to the designer. We show how this freedom provides a spectrum of different design points and how one can take advantage of it to satisfy criteria such as power and area consumption.

First, assume that we use `farmroad_light` objects, which is a good choice since the green-time of lights at the square is fixed, similar to the `farmroad_light` class. The software model is presented in Figure 6, that first defines four objects of `farmroad_light` class (line 2) and then (lines 5 to 8) iteratively opens one of them, in turn, and closes the others (for better readability, a sequence of four commands is shown per line). Software and power consumption results are shown in the second data row of Table 2. The power consumption figures in the third and fourth data columns are produced as in previous section; for the power-down mode, the unused FUs are switched off (the `highway_light::close()` and `traffic_light::open()` FUs in this case). This is an *application-specific* "power management" technique whose effectiveness will be analysed in Section 6.5. In our experiment setup this power-down mode was realised by statically deleting these unused FUs from the Verilog design and then estimating power consumption as before. In case of implementing the OO-ASIP on a Field-Programmable Gate Array (FPGA), this same deleting technique can be used to consume less area; this is an *application-specific* "area management" technique that is enabled by the granularity of the OO-ASIP instructions and its internal architecture. This same deactivation technique can also be applied at run-time to allow *dynamic* power and area management. The MIU is always aware of the FUs that are active at any point in time (because FUs are activated only by the MIU), and hence, the MIU can dynamically switch off inactive FUs. For power management this can be done, for example, by stopping the clock of the inactive FUs, and for area management this can be applied by run-time reconfiguration of the FPGA so that only the *active* FU(s) are implemented in the FPGA. This dynamic deactivation can be applied independent of the application being executed on the OO-ASIP and represents our *dynamic* and *application-independent* power and area management policy. Obviously, these *dynamic* management policies introduce some performance overhead (to switch on or bring into the FPGA the now-invoked FU) that needs to be carefully traded off for the offered benefits.

As mentioned before, one can equally well use `highway_light` objects in this application by simply replacing `farmroad_light` with `highway_light` in

```
1  #include "traffic_lights.h"

2  farmroad_light lights[4];

3  main() {
4   forever do {
5   lights[0].open();    lights[1].close();    lights[2].close();    lights[3].close();
6   lights[0].close();   lights[1].open();     lights[2].close();    lights[3].close();
7   lights[0].close();   lights[1].close();    lights[2].open();     lights[3].close();
8   lights[0].close();   lights[1].close();    lights[2].close();    lights[3].open();
9   }
10 }
                                    main program
```

Figure 6: Software for the four traffic-lights at a square.

line 2 of Figure 6. This results in the figures shown in the third row of Table 2. The power estimation is done as before, but for the power-down mode the `farmroad_light::open()` and `traffic_light::close()` FUs are deleted from the Verilog model.

6.5 Analysis of the Experimental Results

By realising polymorphism in hardware, the dispatching function of the MIU introduces some overhead. However, Table 1 shows that the area overhead is only 6% of the total system area, while taking only one clock cycle. Of course, by incorporating more methods the MIU will be bigger, but the total chip area will also grow; hence, the ratio can remain the same or even decrease as FUs are to be more complicated and area-consuming than the MIU, which merely adds one row to the VMT per new class. When synthesising the chip for higher clock frequencies, the MIU may, or may not, need more clock cycles; this needs more complicated examples to explore. Nevertheless, we have addressed such issues concerning polymorphism overhead in our other work [Goudarzi et al. 04] that dispatches virtual methods at the same time that packets are routed in an NoC platform; this realises polymorphism for free in NoC implementation since no extra hardware is used other than what is already required for packet routing in the NoC.

The OO-ASIP total area in Table 1 seems a little bit high. This is because we did not try to optimise the design and use synthesis-tool options at all. Optimising internals of each FU is the well-known process of "behavioural synthesis" and we do not directly concern ourselves with it. The OO-ASIP structure, the MIU, and the OMU are the concepts that we have introduced and their area overhead is negligible, especially noting that in this preliminary work they have been described at the behavioural level for simplicity. In a real evaluation in silicon, various techniques from mature processor architecture technology will be employed to design them optimally.

In the above case study, we first devised a "hardware class library" for the original problem of highway-farmroad traffic-lights. Then the "hardware class library" and the corresponding OO-ASIP were reused in another similar problem involving four traffic lights at a square. This illustrates how the OO-ASIP is reused. We sadly could not demonstrate "hardware class library" augmentation to illustrate "hardware-accelerator reuse" since it required at least a processor core, if not a compiler, to deal with the "software methods".

Analysis of Power and Area Management Policy. Our choice of the instruction-set and internal architecture of the OO-ASIP enables *statically* purging or turning-off parts of the system that are not used in a certain embedded application. In the first example (highway-farmroad crossing), all class methods were required for system operation. However in the two others, some methods were not used at all; so, they can be statically powered off, or even cut out of the architecture in the case of OO-ASIP realisation on reconfigurable hardware. This offers 'application-specific power and area management' in the OO-ASIP by statically switching off or deleting hardware units that are certainly not used by the application.

Table 2 confirms effectiveness of this policy by showing 15% and 20% reduction in power consumption when using respectively `farmroad_light` and `highway_light` objects in the square lights problem. This reduction is achieved by powering-off (purging, in our experiment setup) the unrequired FUs; i.e. `highway_light::close()` and `traffic_light::open()` in the first case and `farmroad_light::open()` and `traffic_light::close()` in the second one. Similarly, results of OO-ASIP synthesis in Table 1 confirm effectiveness of the area reduction by "method purging": the area can be reduced by 19% and 22% when respectively using `farmroad_light` and `highway_light` objects in square lights problem.

Comparing the second and third data rows of Table 2 shows that the use of `highway_light` objects is preferable due to less power consumption (4 or 10% depending on whether the power-down mode is active or not). The same is true regarding required chip area. This demonstrates another dimension of application-specific static power and area management based on the choice of objects type.

It is worthy of note that the "traffic-light controllers" example elaborated and analysed above is a very simple one that may not benefit highly (and was not intended to) from an object-oriented implementation; it was adopted as a universally well-known digital design example that firstly is simple enough to allow its OO model be easily understood by general readers, and secondly allows concentration on the synthesis and reuse methodology instead of the modelling methodology that we wish not to deal with in this paper. Instead, the interested reader is referred to [Wolf 01] where object-oriented models are presented for several applications in a variety of domains, ranging from as simple as an alarm clock to as complex as a video-accelerator. As already mentioned in Section 2.3, different types of systems may benefit from different system-level models and design styles. Our methodology is aimed at general consumer electronics systems, however, elaborate characterisation of the type of systems that would benefit from modelling and implementation in our methodology is an essential part of our further research.

7 Comparison to Related Work

Several previous works address OO hardware synthesis by extending VHDL [Radetzki 00, Ashenden et al. 97, Schumacher and Nebel 95], synthesis from Java [Kuhn et al. 01, Young et al. 98], and from SystemC models [Grimpe et al. 02, Schulz-Key et al. 04]; however, none of them involve ASIP approach.

Radetzki [Radetzki 00] proposes to synthesise objects as finite-state machines. The object data fields comprise the object state. The methods, each implemented as a hardware module, manipulate these state bits. He uses some bits in the object storage to show its class membership and hence, his objects can dynamically change type. To implement polymorphism, each object implements in hardware all the methods that a virtual method may be dispatched to. When a virtual method is called, all potential implementations are activated and a multiplexer at their output ports selects the output value of the only implementation that corresponds to the run-time class membership of the polymorphic-object. The expansion of many methods for every object carries a significant area cost. Moreover, the polymorphism realisation approach introduces unnecessary power consumption due to activating many implementations but using only one of them. These are addressed in our ODYSSEY solution. Radetzki's approach has also been used in the European ODETTE project [ODETTE 03], which introduces SystemC-Plus as an extension to SystemC with synthesisable object-oriented features [Grimpe et al. 02]. Both Radetzki and ODETTE aim at synthesising an ASIC from the OO model and hence do not talk about post-manufacturing extensions of the OO model and hardware reuse through programmability.

In the OASE project [OASE 03], Kuhn et al. transform an OO specification to a non-OO specification and then use behavioural synthesis tools to synthesise it into hardware [Kuhn et al. 01, Schulz-Key et al. 04]. They implement polymorphism as a switch-case statement to test the object type at run-time and call the appropriate method, which again causes significant area overhead as they have already reported in [Kuhn et al. 01]. Current OASE work is also limited to ASIC synthesis and does not address programmability.

Parvataneni et al. at Silicon Infusion Ltd. propose an object-orientated heterogeneous multiprocessor platform [Parvataneni et al. 03], which uses a network structure to dispatch messages (method invocation commands) among processing elements that are either hardware modules or normal processors. They propose to use *firmware* to dynamically resolve virtual methods to the appropriate processing elements [Parvataneni and Nanetti 03], and hence, they would also allow overriding hardware by software. Our work is very close to theirs especially regarding the internal architecture; however, we follow a top-down design flow while they start from IP blocks for the processing elements and follow a bottom-up approach.

The "class library" of our OO-ASIP is devised by the system designer effectively capturing his experience and insight on target applications. Hence, our instruction selection approach reflects a direct hint from the designer. Similarly, Wolf [Wolf 96] has used designer-provided decomposition hints (objects and methods) in OO specifications to partition an OO model and co-synthesise it to a distributed engine of heterogeneous processors and their software. However, he assumes the processors are given and does not synthesise them.

Each of the above approaches has its own merits; however, to the best of our knowledge, our ODYSSEY approach is the first in the literature that proposes an ASIP approach to implementing OO models, that enables hardware/software co-design from an OO model while proposing "class methods" (in contrast to a "complete object") as the partitioning quantum, and that realises polymorphism in an ASIP.

8 Summary and Conclusion

The main thrusts of this paper are firstly to introduce our design flow for embedded system development based on employing OO-ASIPs, secondly to present a policy of hardware/software co-design where the hardware and software components are both specified in the same language and even the same syntax, and finally to demonstrate the reuse methodology and power/area management policies that the employment of OO-ASIPs provides.

We proposed a methodology for embedded system design persuading object-oriented design of applications at system level to run on the OO-ASIP that is tailored to the class library corresponding to the application domain. The issues on synthesising and programming the OO-ASIP were discussed and the practical use of the methodology was shown by some case studies. Experimental results of implementing the case studies were presented and analysed to demonstrate the advantages of the approach in ASIP reuse, in application-independent power and area management (i.e. dynamic deactivation of unused parts), and in application-specific power and area management (i.e. static deactivation of unused parts).

We have presented a prototype internal architecture for the OO-ASIP. However, the core point on which we would like to emphasise is the OO-ASIP instruction-set, not the architecture. Other architectures can be proposed employing prior knowledge of processor design or innovative solutions. Indeed we have proposed a variant architecture [Goudarzi et al. 04] that realises polymorphism for free. We view at least the following areas for further research and for revisiting in order to suggest customised innovative solutions for the OO-ASIP: alternative architectures to address instruction-level parallelism or multi-threading, memory architectures, and caching policies and mechanisms.

We intend to use the same language, C++, over all development stages by moving from Verilog to SystemC for OO-ASIP synthesis. Currently we are focusing on the networked internal architecture mentioned above. A retargettable compiler will then be developed to accelerate application development and implementation of case studies.

Acknowledgements

We wish to thank the British Council for funding the first author's visit to Cambridge, and also a partial scholarship from the Ministry of Science, Research, and Technology of the Islamic Republic of Iran. This work was partly supported by (UK) EPSRC grant GR/N64256.

References

[Ashenden et al. 97] Ashenden, P.J., Wilsey, P.A., Martin, D.E.: "SUAVE: painless extension for an object-oriented VHDL"; Proc. VHDL International Users Forum (VIUF, Fall Conference), (1997).

[August et al. 02] August, D.I., Keutzer, K., Malik, S., Newton, A.R.: "A disciplined approach to the development of platform architectures"; Microelectronics Journal, Elsevier, (2002).

[Benini and DeMicheli 02] Benini, L., DeMicheli, G.: "Networks on chips: a new SoC paradigm"; IEEE Computer, 35, 1 (2002), 70-78.

[Booch 94] Booch, G.: "Object-oriented analysis and design with applications (2nd edition)"; Addison-Wesley, (1994).

[Cooling 03] Cooling, N.: "UML and its role in real-time embedded systems design"; IEE one-day seminar on Developing Embedded Real-Time Systems, IEE Embedded and Real-Time Systems Professional Network, London (2003).

[Goudarzi et al. 03] Goudarzi, M., Hessabi, S., Mycroft, A.: "Object-oriented ASIP design and synthesis"; Proc. Forum on specification & Design Languages (FDL'03), Frankfurt (2003).

[Goudarzi et al. 04] Goudarzi, M., Hessabi, S., Mycroft, A.: "Overhead-free polymorphism in network-on-chip implementation of object-oriented models"; Proc. Design Automation and Test in Europe (DATE'04), Paris (2004).

[Grimpe et al. 02] Grimpe, E., Timmermann, B., Fandrey, T., Biniasch, R., Oppenheimer F.: "SystemC object-oriented extensions and synthesis features"; Proc. Forum on specification & Design Languages (FDL'02), (2002).

[ITRS 01] International Technology Roadmap for Semiconductors (ITRS)–Design, `http://public.itrs.net/Files/2002Update/2001ITRS/Design.pdf`, (2001).

[Keutzer et al. 02] Keutzer, K., Malik, S., Newton, A. R.: "From ASIC to ASIP: the next design discontinuity"; Proc. Int'l Conference on Computer Design (ICCD'02), (2002).

[Kuhn et al. 01] Kuhn, T., Oppold, T., Winterholer, M., Rosenstiel, W., Edwards, M., Kashai, Y.: "A framework for object oriented hardware specification, verification, and synthesis"; Proc. Design Automation Conference (DAC'01), (2001).

[Lee and Tepfenhart 97] Lee, R.C., Tepfenhart, W.M.: "UML and C++: A practical guide to object-oriented development"; Prentice-Hall, (1997).

[MESCAL 03] The MESCAL Project: Modern Embedded Systems, Compilers, Architectures, and Languages, `http://www.gigascale.org/mescal/`

[OASE 03] The OASE Project: Objektorientierter hArdware/Software Entwurf, `http://www-ti.informatik.uni-tuebingen.de/~oase/`

[ODETTE 03] The ODETTE Project: Object-oriented co-DEsign and functional Test Techniques, `http://odette.offis.de`

[ODYSSEY 03] The ODYSSEY Project: Object-oriented Design and sYntheSiS of Embedded Systems, `http://ce.sharif.edu/~odyssey/`

[Parvataneni et al. 03] Parvataneni, T., Nanetti, G., Holgate, C., Eland, H., Onions, P., Wray, F.: "Object-orientated heterogeneous multiprocessor platform", European and US patent application (GB2381336), `http://l2.espacenet.com/espacenet/viewer?PN=GB2381336`, (2003).

[Parvataneni and Nanetti 03] Parvataneni, T., Nanetti, G.: Private communication in The Computer Laboratory, University of Cambridge, June 20th, (2003).

[Radetzki 00] Radetzki, M.: "Synthesis of digital circuits from object-oriented specifications"; PhD Thesis, University of Oldenburg, (2000).

[Schulz-Key et al. 04] Schulz-Key, C., Winterholer, M., Schweizer, T., Kuhn, T., Rosenstiel, W.: "Object-oriented modeling and synthesis of SystemC specifications"; Proc.the Asia South Pacific Design Automation Conference (ASPDAC'04), Yokohama (2004), 238-243.

[Schumacher and Nebel 95] Schumacher, G., Nebel, W.: "Inheritance concept for signals in object-oriented extensions to VHDL"; Proc. EURO-DAC with EURO-VHDL, (1995).

[Sgroi et al. 00] Sgroi, M., Lavagno, L., Sangiovanni-Vincentelli, A.: "Formal models for embedded system design"; IEEE Design & Test of Computers, 17, 2 (2000), 14-27.

[SystemC 04] The SystemC Initiative, `http://www.systemc.org`

[SystemVerilog 04] The SystemVerilog Homepage, `http://www.systemverilog.org`

[Tsai et al. 02] Tsai, M., Kulkarni, C., Sauer, C., Shah, N., Keutzer, K.: "A benchmarking methodology for network processors"; 1st Network Processor Workshop, Proc. Int'l Symposium on High Performance Computer Architecture (HPCA), Boston, MA (2002).

[Wolf 96] Wolf, W.: "Object-oriented co-synthesis of distributed embedded systems"; ACM Transactions on Design Automation of Electronic Systems (TODAES), (1996).

[Wolf 01] Wolf, W.: "Computers as components: principles of embedded computing system design"; Morgan Kaufmann Publishers, (2001).

[Young et al. 98] Young, J.S., MacDonald, J., Shilman, M., Tabbara, A., Hilfinger, P., Newton, A.R.: "Design and specification of embedded systems in Java using successive, formal refinement"; Proc. Design Automation Conference (DAC'98), (1998).

A Details of Traffic-Light Controller OO-ASIP Synthesis

In this appendix we present the details of synthesising an OO-ASIP for the case study discussed earlier in Section 6. For this simple example, one would suggest the class library of Figure 4 rooted at `traffic_light` class. This class includes a `state` data member (whether light is in red, green, or yellow state) and an `elapsed_time` data member (how long this state has been active), and three methods `open()`, `close()`, and `timekeeper()`, which turn the light to green, turn it to red, and keep track of time units, respectively. There are two variants of this class: farmroad and highway lights. Hence, two classes are derived: `farmroad_light` and `highway_light`. The former adds a new data member named `fixed_green` and specialises the `open()` method accordingly. The latter adds the new `min_green` data member and overrides the `close()` method. Details of the class library and methods are also presented in Figure 4 and discussed in Section 6.2.

A.1 Synthesising the *Traffic-Light Controller* OO-ASIP

In our experiment setup, the OO-ASIP is described in Verilog HDL, simulated, and then synthesised. Five class methods exist in the class library rooted at `traffic_light`. As an example of our FU synthesis procedure, Verilog code of `open()` method of the `traffic_light` class, i.e. the `traffic_light$open` FU, is provided in Figure 7 (we named each `class::method` FU as `class$method` module in Verilog for better readability and correct Verilog syntax). In lines 1 to 6 the Verilog *module* and its ports and parameters are defined. Then from line 8 to 45 Verilog tasks are defined that are used to contact the OMU to access object fields. The `read` task is discussed here in detail; all other task operate similarly. The `read` task is to read from data memory the value of an object field designated by a virtual-address (*oid*, *index*), where *oid* is the object identifier and the *index* is the index of the requested field in the object data storage. The `oid` is an input port of the Verilog module (line 1) and hence is already available to the `read` task. The `index` is specified as an input to the task (line 11). The output of the task is the data read from the data memory (through the OMU) and is specified as `d` in line 10. First, the virtual-address is formed in line 13. Then, four control signals of the OMU are assigned appropriate values in line 14; the `rd`, `wr`, `lock`, and `unlock` control signals respectively designate a read, write, lock, and unlock request to the OMU. Line 15 waits for the `ready` signal from the OMU that shows availability of the requested data, and then accordingly assigns the task output value (i.e. the `d` argument of the task). Finally line 16 returns the OMU control signals to the inactive state. As can be seen in Figure 7, for each operation (i.e. `read` or `write`) there are three variants; e.g. `read` (lines 9 to 18), `read_lock` (lines 20 to 29), and `read_unlock` (lines 31 to 40). The *locking* version asks the OMU to block other access requests to that object field until an *unlocking* operation is done by this same FU; this facilitates atomic update.

Finally, lines 47 to 61 define the FU functionality in behavioural Verilog. This involves re-specifying each method functionality in Verilog using the above-

```
1 module traffic_light$open( clk, reset, oid,
2    start, done, vaddr, rd, wr, lock, unlock,
3    ready, data_in, data_out);
4
5 // parameters and ports definitions
6  ...
7
8 // Task definitions
9 task read;
10   output [data_width-1:0] d;
11   input[max_index_width-1:0] index;
12 begin
13   vaddr={oid, index};
14   rd=1; wr=0; lock=0; unlock=0;
15   wait(ready) d=data_in;
16   rd=0;
17 end
18 endtask //read
19
20 task read_lock;
21   output [data_width-1:0] d;
22   input[max_index_width-1:0] index;
23 begin
24   vaddr={oid, index};
25   rd=1; wr=0; lock=1; unlock=0;
26   wait(ready) d=data_in;
27   rd=0; lock=0;
28 end
29 endtask //read_lock
30
```

continued in the right panel

continued from the left panel

```
31 task read_unlock;
32   output [data_width-1:0] d;
33   input[max_index_width-1:0] index;
34 begin
35   vaddr={oid, index};
36   rd=1; wr=0; lock=0; unlock=1;
37   wait(ready) d=data_in;
38   rd=0; unlock=0;
39 end
40 endtask //read_unlock
41
42 // Other tasks similar to the previous ones
43 task write; ...
44 task write_lock; ...
45 task write_unlock; ...
46
47 reg done;
48 reg [7:0] state;
49 parameter state_index=0;
50
51 always @(start or reset) begin
52   if (reset)
53     done = 1;
54   else if (start) begin
55     done = 0;
56     read(state, state_index);
57    if (state!=GREEN)
58      update_state(GREEN);
59     done = 1;
60   end // else if
61 end // always
62
63 endmodule //traffic_light$open
```

traffic_light::open() method

Figure 7: Verilog code for `traffic_light$open` FU. Bold italic text shows part of the template of OO-ASIP architecture. Other parts come from the method definition.

mentioned Verilog tasks for simulation and synthesis. This part of the FU is specified in normal font in Figure 7. The bold italic portions of the figure are the FU- and application-invariant templates that define the OO-ASIP architecture. All other FUs are defined in the same template.

Comparing lines 56, 57, and 58 of Figure 7 to the C++ definition of the `traffic_light::open()` method in Figure 4 (top-right panel, first method definition) confirms the high resemblance of the FU behavioural code to the method definition. All other code in lines 47 to 55, and 59 to 61 are provided just to allow the MIU to control and monitor the operation of the FU. The high resemblance between the method definition and the FU behavioural code shows possibility of designing a translation tool that generates FU modules from the code of class methods. This is part of our current work in providing an automated synthesis tool framework for our methodology.

For time-keeping of the lights, an internal timer in the OO-ASIP issues interrupts and the corresponding interrupt service routine updates the `elapsed_time` attribute of all `traffic_light` objects in the system (see Section 6.3). This is an example of concurrent attribute access in system operation and hence requires a control mechanism for shared access. Figure 8 shows the use of `read_lock` (line

```
1 module traffic_light$timekeeper( clk, reset, oid,
2    start, done, vaddr, rd, wr, lock, unlock,
3    ready, data_in, data_out);
4
5 // parameters and ports definitions
6 ...
7
8 // Task definitions
9 task read;
10   output [data_width-1:0] d;
11   input[max_index_width-1:0] index;
12 begin
13  vaddr={oid, index};
14  rd=1; wr=0; lock=0; unlock=0;
15  wait(ready) d=data_in;
16  rd=0;
17 end
18 endtask //read
19
20 task read_lock;
21   output [data_width-1:0] d;
22   input[max_index_width-1:0] index;
23 begin
24   vaddr={oid, index};
25   rd=1; wr=0; lock=1; unlock=0;
26   wait(ready) d=data_in;
27   rd=0; lock=0;
28 end
29 endtask //read_lock
30
```

continued in the right panel

continued from the left panel

```
31 task read_unlock;
32   output [data_width-1:0] d;
33   input[max_index_width-1:0] index;
34 begin
35   vaddr={oid, index};
36   rd=1; wr=0; lock=0; unlock=1;
37   wait(ready) d=data_in;
38   rd=0; unlock=0;
39 end
40 endtask //read_unlock
41
42 // Other tasks similar to the previous ones
43 task write; ...
44 task write_lock; ...
45 task write_unlock; ...
46
47 reg done;
48 reg [7:0] state;
49 parameter state_index=0;
50
51 always @(start or reset) begin
52  if (reset)
53    done = 1;
54  else if (start) begin
55     done = 0;
56     read_lock(elapsed_time, elapsed_time_index);
57     elapsed_time = elapsed_time + 1;
58     write_unlock(elapsed_time, elapsed_time_index);
59     done = 1;
60   end // if
61 end
62
63 endmodule //traffic_light$timekeeper
```

traffic_light::timekeeper() method

Figure 8: Verilog code for `traffic_light$timekeeper` FU. Bold italic text shows part of the template of OO-ASIP architecture. Other parts come from the method definition.

56) and `write_unlock` (line 58) Verilog tasks supported by the OMU to facilitate atomic updates. The `read_lock` Verilog task reads the `elapsed_time` variable and asks the OMU to lock it, then the `write_unlock` Verilog task updates that object fields and asks the OMU to release the lock. As the figure shows, the FU code in bold-italic font is FU-invariant and is just the same as in Figure 7.

The entire model was simulated and validated using ModelSim© commercial HDL simulator. The initial model, using Verilog tasks as presented above, was not directly synthesisable by LeonardoSpectrum© commercial synthesis tool. This was due to the use of timing constructs in the task definitions of the FUs and the MIU, and also because of concurrent access to variables of the Verilog module in the OMU specification. We inlined the tasks and integrated all of the concurrent accesses into a single `always` block to synthesise them. The area/delay results of synthesis over a sample 0.5 μm process technology are shown in Table 1.