Acknowledgements

The ideas presented in this thesis have been developed while I was working for the EU projects REQUEST (no. 20616) and SQUASH (no. 28889).

I am grateful to my supervisor, Prof. Dr. Wolfgang Nebel, for enabling this research and supporting my work.

I would like to thank Prof. Dr. Wolfgang Rosenstiel for the effort he spent in examining this dissertation.

My colleagues, Dr. Guido Schumacher and Wolfram Putzke-Röming, have reviewed the manuscript. Thanks again.

Many thanks also to all individuals and companies who have shared their opinion, suggestions, and other input (such as LEDA's Objective VHDL analyser) in the framework of the before-mentioned projects.

Not least, Ansgar Stammermann and Tiemo Fandrey have made valuable contributions to the implementation of my ideas.

Chapter 1

Introduction

Digital hardware is becoming increasingly complex at an exponential rate. The number of primitive logic gates of which a processor is built has been reported to approximately double every 18 months for a long time (Moore's law). Today, designs comprising tens of millions of transistors can be manufactured. In the Semiconductors Industry Association (SIA) roadmap [148], it is foreseen that the exponential growth can be sustained over the next decade, physical limits of digital electronics being still not in sight. The technical progress will enable the production of devices with sophisticated functionality at an affordable price and the integration of complete systems on a single, miniature format chip instead of a printed circuit board.

To utilise that potential, our capabilities of designing electronic systems must keep pace with the technology. Today, however, design productivity rises at a much lower rate than necessary, opening the so-called design gap. Since it is not feasible to scale up design teams indefinitely, future design methods must provide a major productivity leap. This is likely to require a paradigm shift. Hardware design will have to happen at a much higher level of abstraction. Tasks that are performed manually today will have to become automised. Reuse of previously designed components will have to take place at a large scale and in an organised way.

Time-to-market is another pressing issue. A new or enhanced product must become available to its potential consumers within a time interval known as the market window. Premature and late market introduction both incur a cost of lost profit. With product cycles shortening, the timely delivery of a design becomes more and more important. Again, productivity increases contribute to meeting this goal. Moreover, late design iterations must be avoided. A design iteration is necessary when an error is detected. Often this happens only after system integration since the system components, particu-

larly software and hardware, are designed independent of another despite their complex interactions. A more integrated and co-operative design style and the reuse of tested components can help to reduce errors and meet time-to-market requirements.

In the past, language-based design techniques from the software world have already been adopted into hardware design methodologies. Hardware description languages have largely replaced schematic-based design. The software world, however, is one step ahead in that it widely employs object-oriented techniques. These have been developed since the early 1980s as a response to massive problems with software complexity and defects; problems similar to those faced in hardware design today.

It is expected that by utilising object-orientation in hardware design, we can benefit from its abstraction and reuse techniques. Moreover, object-orientation would then have the potential to unify software and hardware design. To this end, a design methodology and a complete design flow supported by tools must be developed. The work at hand makes a significant contribution to this overall goal by investigating the synthesis of digital circuits from object-oriented specifications.

Our goal is to provide a wide support for object-oriented features and their use in concurrent hardware models while giving designers an idea of the circuit structures that will be synthesized. Analysis and optimization techniques shall be developed so as to reduce implementation inefficiencies that might be a penalty of the more abstract, object-oriented description style. Moreover, by generating standard hardware description language code from a model described in an object-oriented language, we target at providing a viable transition from the object-oriented world to current hardware design flows.

The next chapter provides an overview of digital system design and describes how object-orientation can contribute to it. This background is used in chapter 3 to illustrate and classify existing approaches to hardware design and synthesis based on object-oriented languages, and to distinguish our contribution from the related work.

Chapter 4 presents object-orientation in detail, following the Unified Modeling Language, a de facto standard for the diagrammatic specification of object-oriented models. This allows to keep our view independent of a specific implementation language. We devise a hardware interpretation and implementation for all major object-oriented features and identify some necessary restrictions on their use.

The following three chapters target at making object-oriented hardware synthesis practicable. In chapter 5, the language Objective VHDL and its application for specifying object-oriented synthesis models are presented. Chapter 6 addresses analyses that enable an optimized hardware implementation, concentrating on the aspects related to object-orientation. In chapter 7, we describe the generation of VHDL code to enable the interfacing with existing design flows and tools.

Chapter 8 is devoted to design experiments that demonstrate and validate our concepts and help to identify remaining weaknesses, leading to conclusions summarising the contribution of this thesis and future work necessary.

Chapter 2

Design of Digital Systems

This chapter provides relevant background on the industrial and research state of the art in digital systems design. We briefly review models of the design space and identify important design activities. Hardware design methodologies currently in use in industry or subject of research are investigated. Likewise, we highlight the predominant software engineering methodology, object-orientation.

The advantages of using object-orientation are expected to be transferrable from the software domain to systems design, including hardware design, as we will point out. Object-orientation could become a unified methodology for the specification of software and hardware. To this end, however, we will need support for implementing hardware that is specified in an object-oriented way. An object-oriented language applicable to hardware description must be found or developed. Automated support for the generation of a hardware implementation from a language based description must be provided and should interface with present hardware design flows.

2.1 Design space

The design space of a digital system can be understood as the collection of the views and abstraction levels relevant in the process of designing it. Several approaches to capturing the digital system design space exist. These include Gajski's Y-chart, Ecker's design cube, and the VSI taxonomy.

The Y-chart is discussed in its refined form presented in [46] and [149]. It describes the hardware design space as spanned by three dimensions or views, behaviour, structure, and geometry, which are represented by axes arranged in a two-dimensional Y shape. On each axis, there are discrete points that correspond to abstraction levels, namely, from the centre to the

outside, the switch, gate, register-transfer, algorithmic, and system level. Concentric circles connecting the different views at the same level can be drawn. Rammig [134] has refined the Y-chart into an X-chart by adding a test view.

Ecker [46] has developed a design space characterisation that is particularly suited for the design process of application specific integrated circuits (ASICs) using hardware description languages (HDLs). Axes represent the ways functionality, data, and timing are described. Functionality can be modelled with a structural, concurrent, or sequential description style. Data can be represented by single bits, bit-vectors, or abstract data types. Timing may be a delay measured in physical units, related to clock cycles, or expressed by causal relationships. Since these axes are orthogonal to each other, they are arranged in a three-dimensional way. The 27 combinations of axis values are represented as 27 points in the design cube.

The VSI taxonomy [163], similar to the RASSP taxonomy [135], targets at taking not only hardware, but also software aspects into account. To this end, it defines the orthogonal model characteristics of temporal, data value, functional, and structural precision, for each of which an internal (implementation) and external (interface) view exist. A ninth dimension, programming abstraction level, accounts for the software aspect. The axes are drawn one below the other. A region of the design space is represented by a combination of intervals on these axes.

2.2 Design activities

A design activity or design step is an action performed by a designer, possibly with tool support, in the process of designing a digital system. Design activities relevant to this work are modelling, exploration, synthesis, optimization, and validation. We will describe these activities in the following and relate them to the design space.

By *modelling*, a view of a design is described at a level of abstraction suitable for a given purpose, leaving out those details of the reality that are irrelevant. For instance, a behavioural model at the algorithmic level (in the Y-chart) can be suitable to express a desired functionality whereas a more detailed geometric model at the physical level may be required to capture the design data that is needed for the production of an integrated circuit.

Synthesis is the process of generating a model at a lower level of abstraction, called *implementation*, from a higher level model called *specification*.

In hardware synthesis, the specification is typically a behavioural model at some level of abstraction in the Y-chart, and the implementation is a structural model whose components are described one abstraction level lower. While synthesis is controlled by a designer by providing constraints and directives, large parts of this design activity are performed by tools. Hence, the process of detailing a specification into an implementation is largely automated, which helps to increase design productivity and reduce design time. If the designer must perform this refinement manually, we speak of (manual) implementation.

Closely related to synthesis are optimization, exploration, and estimation. When refining a specification into an implementation, there are typically many alternatives among which a choice must be made. This choice affects the implementation's efficiency with respect to hardware area, software code size, performance, power dissipation, and testability, just to name the most important criteria. *Optimization* means to make choices so that the implementation represents a good trade-off between these interrelated criteria, compared to other possible implementations. In many cases, this cannot be assessed in advance, so that the space of possible implementations must be *explored*. However, complexity usually forbids to generate and analyse more than a few alternative implementations, making it necessary to apply approximative techniques (*estimation*).

Another important design activity is to validate that an implementation behaves according to its specification. *Validation* deals not only with functionality, but also timing and performance aspects. The most important validation technique has been to simulate the specification and implementation models and compare the results. More recently, formal verification in the sense of proving model equivalence has become popular. Other techniques, model checking or theorem proving, can be applied to show that a model fulfils properties specified in a mathematical notation.

2.3 Design methodologies

A design methodology describes how to get from an initial specification to a final implementation by systematically carrying out design activities. The industrial state of the art in designing hardware and embedded systems is outlined in this section and compared with advanced approaches currently being researched.

2.3.1 Hardware design

Industrial hardware design today is largely based on the use of HDLs for modelling and synthesis from register transfer level (RTL) models [26]. The most prominent and widely used HDLs are VHDL [74] and Verilog [157]. RTL models are characterised by describing functionality as separate datapaths and controllers. A behavioural view is taken for modelling each datapath and controller, describing their functionality using a synthesizable subset of HDL statements [75]. The classification of RTL modelling in the Ecker cube is instructive: All timing is described in relation to a clock signal, and data are typically bits and bit-vectors. The synthesis tools fully automate the generation of a gate level netlist, yielding acceptable results in a time much shorter than achievable with manual implementation. Further design steps towards layout, such as placement and routing, are automated, too.

Research has taken the entry point to synthesis one abstraction level further, to behavioural models at the algorithmic level [42][102]. At this level, hardware functionality is modelled as a set of sequential processes. No separation between control and data path is made in the specification. In the design cube, the timing description can be classified as causal, being defined by the order of statement execution. Data are typically modelled as bit-vectors or with primitive data types such as integers.

An implementation is generated from an algorithmic description by so-called high level synthesis (HLS) tools, also known as behavioural synthesis tools. A HLS tool takes a sequential algorithm and schedules its operations into time slots that correspond to clock cycles. It allocates modules that implement the operations required, and binds operations to these resources. The interconnection of resources according to the data flow yields a data path. Furthermore, a finite state machine is generated after the schedule to control the data path. The resulting implementation is suitable as a specification for RTL synthesis.

Constraints play an important role to guide the HLS tool. Timing constrained and resource constrained synthesis algorithms are distinguished [79]. Timing constrained scheduling attempts to minimize resources while observing constraints on the algorithm's execution time (latency). Resource constrained scheduling minimizes the execution time of an algorithm given a pre-allocated set of resources.

Commercial HLS tools have become available [84]. They are, however, not as mature and widely accepted in industry as RTL synthesis tools. The synthesis results cannot compete with a manual RTL design in many cases.

On the other hand, by automating scheduling, allocation, and binding, HLS allows to explore the RTL implementation space much more effectively than manual RTL design. With growing design complexity and time-to-market pressure, and further advances in the tool world, a wider acceptance of HLS techniques can be anticipated.

Contemporary research targets system level modelling and synthesis [80], dealing with hardware architectures that include microprocessors and microcontrollers, digital signal processors (DSPs), application specific instruction set processors (ASIPs), application specific integrated circuits (ASICs), and memory hierarchies. Interfacing of and communication between these building blocks must be addressed. The creation, selection, and reuse of intellectual property (IP) blocks is another important issue. Due to the consideration of programmable processors, software comes into play at the system level.

2.3.2 System design

We use the term system to denote a unit that is comprised of software and digital hardware. Other system components such as analog electronics and mechanical parts are out of scope for our considerations.

In industry, such hardware/software or embedded systems are designed using separate design methodologies for their software and hardware parts. System functionality and constraints are specified in an informal way. Thereafter, a partitioning decision is made on the system's components to implement them in hardware and software, respectively. Until recently, a system integration and test took place only after the software had been implemented and the hardware been produced. Any errors in their interplay could not be detected before this final phase and would in many cases require a large redesign effort, causing significant delay and cost [4].

In the past years, techniques for the (non-formal) co-verification of hardware and software have been developed and were adopted into industrial design practice. These include the co-simulation of hardware components together with software, rapid prototyping of hardware to allow an earlier integration, and the employment of emulation techniques. However, while earlier debugging has been enabled, other design activities continue to be disconnected.

Research addresses the co-operative design (co-design) of hardware and software using an integrated methodology [50]. The heterogeneous approach to co-design leaves traditional means for hardware and software design in use, e.g. HDLs and high-level programming languages (HLLs), while pro-

viding a tighter coupling through an underlying integrated semantic model that facilitates the automation of cross-domain design activities. The Solar intermediate form [78] is an example of a multi-language co-design environment. The homogeneous approach, on the other hand, targets at making the specification independent of an implementation in either hardware and software by using a single language for both [56][114]. This allows to leave the partitioning decision and further co-synthesis activities to an automated tool such as the Cosyma system [48].

Given that software design is the dominant effort, accounting for an average of about 80% of a system's engineering costs [8], it is reasonable to adapt a software engineering methodology to hardware design when striving for homogeneous system design. Among the software techniques, the object-oriented paradigm represents an important, if not the most dominant, methodology. Its application to system design has been suggested by several researchers [67][109].

2.4 Object-oriented software engineering

Object-orientation relies on a couple of features for the abstraction of data, functionality, and communication, which we present briefly in the following. A more detailed examination will be performed in chapter 4, with a hardware implementation in mind. In addition to features, we mention the software development process describing their use for analysing, designing, and programming software, as an integral part of an object-oriented methodology. A special branch of object-orientation, concurrent object-oriented programming, deals with a concurrent implementation of object-oriented models. Since hardware is inherently parallel, the problems discovered in this context are of high relevance to this work.

2.4.1 Notations and features

The *object* is the fundamental unit on which the object-oriented paradigm is based. An object models a part of a system or of the system's environment. For this purpose, it has the capability of storing and processing information. It defines an interface of services (operations, methods) that it provides, while encapsulating its internal information and hiding implementation details.

Object-orientation allows to classify similar objects. A *class* defines the common properties of a set of objects, particularly its services and internal

data storage capabilities. Objects can be created according to the blueprint described by a class. The *inheritance* of classes enables a designer to extend the class for additional data and services and to redefine a service so as to modify or adapt its functionality.

Message passing enables communication between objects. Messages correspond to services. By addressing an object and sending it a message, an object can requests the execution of a service from the target object.

Polymorphism allows to deal in a uniform way with objects of classes that are related by inheritance. It allows to address an object regardless of its exact class membership and to send it a message. The message is dispatched to the corresponding service. Recall that the service's functionality may be redefined in derived classes. Since the exact class of the addressed object may vary during system operation, it must be decided dynamically which service version to execute. This is achieved by a mechanism known as *dynamic binding* (late binding).

In our terminology, following Booch [28], all these features must be present for a methodology to be object-oriented. In accordance with Wegner [161], approaches that provide objects, classification, and message passing, but lack inheritance or polymorphism, are classified as object-based.

2.4.2 Design process

An important aspect of object-orientation is that it covers not only implementation or programming, but also addresses and formalises preceding design activities. According to Rumbaugh [137], object-oriented modelling consists of three main phases. The first one is object-oriented analysis (OOA), during which requirements are collected, potential objects are identified, and their interrelations such as classification, inheritance, and composition, are determined. Analysis is followed by object-oriented design (OOD) which covers the functional aspects such as data and control flow. Finally, object-oriented programming (OOP) allows to implement the system using an object-oriented programming language.

The object-oriented features are consistently available during analysis and design and in the programming phase. In the last years, the Unified Modeling Language (UML) [29][54][138], which has emerged from various other OOA/D notations [27][76][137], has become a de facto standard for the graphical representation of OOA/D specification models. It includes diagrammatic notations for objects, classes, inheritance, and composition. Beyond these static properties, UML can represent a system's dynamics such

as polymorphism, message exchange, data flow, and control. A specific design process based on the UML, called Unified Process, has been developed recently [77]. For the implementation of object-oriented models, there exist many object-oriented programming languages, including C++ [151], Java [7], and Ada 95 [22].

Tool support is an important factor in the design process. There exist computer-aided software engineering (CASE) tools that allow to capture an OOA/D model. Many of these provide OOP language code generation so that implementation can be partly automated. Furthermore, some CASE tools have reengineering capabilities, allowing to create or modify an OOD model from an OOP implementation. Thereby, the OOD model can be kept consistent with changes introduced during programming. Finally, the optimized compilation of OOP languages into machine code for execution on a processor or virtual machine is a fully automated and proven technology.

2.4.3 Object-orientation and concurrency

An OOA/D model does typically not preclude concurrency. Objects can in principle operate concurrently with respect to each other, using message passing for communication and synchronisation. However, many OOP languages, e.g. C++ and Eiffel [106], provide a sequential execution of object-oriented programs. Yet there are approaches to concurrent object-oriented programming (COOP). These should be taken into account for object-oriented system design since components at the system level, particularly hardware components, are of a parallel nature.

Important issues that arise when combining object-orientation and concurrency are described in [103]. A particular research topic has been synchronisation, of which two forms are relevant. First, *mutual exclusion synchronisation* must be provided in order to avoid resource conflicts between concurrent services of an object. Resources are, for instance, the object's data stores, whose integrity could be compromised by an uncontrolled access by concurrent services. Second, *condition synchronisation* is necessary to avoid the invocation of a service that the object cannot execute temporarily, while enabling other concurrent service requests. For instance, a DRAM controller cannot provide a read or write service when it is in a refresh state.

The problem is that any code that describes synchronisation is prone to becoming invalid during inheritance, having to be re-written in a derived class. Moreover, due to so-called *feature interaction* even the inherited meth-

ods may be compromised if they interfer with synchronisation via the object's internal state. Making it necessary to fully analyse the implementation of a parent class before making any derivations, these effects, subsumed under the term *inheritance anomaly*, severely damage object-oriented encapsulation and reuse.

Major COOP languages, such as Java and Ada 95, provide constructs for mutual exclusion synchronisation: synchronised methods and protected objects, respectively. They do, however, not give the user any support for condition synchronisation in combination with inheritance. COOP languages from research have targeted at *reflective modelling*, giving the programmer the means to describe the acceptance of messages and invocation of corresponding services explicitly in a so-called body of an active object [112]. Whereas this does not prevent the user from running into inheritance anomalies, other approaches based on *guarded methods* can avoid the known issues. This concept, originally described by Ferenczi [53], has been taken further and applied to hardware modelling by Schumacher [144].

2.5 Object-oriented systems engineering

Given the various aspects of object-orientation, we motivate in this section the advantages expected from applying object-oriented techniques to systems design. Initial approaches in this direction are investigated with attention to their support for the object-oriented implementation of the hardware part. As we will see, support for synthesis from object-oriented models is necessary for an effective integration with established hardware design techniques. This leads us to the classification of object-orientation in the digital circuit design space and its integration into an object-oriented system design flow.

2.5.1 Advantages of object-orientation

The potential benefit from the use of object-oriented techniques in hardware/software co-design has been highlighted by several authors, including Wolf et al. [92] and Glunz et al. [59]. Gupta [63] mentions the need for data and interface abstraction in systems design. It is expected that the advantages identified in software engineering can be transferred to system design, including hardware design. These are:

- Abstraction—Object-orientation provides abstract interfaces constituted of methods, the abstract communication mechanism of message passing, and data abstraction.

- Encapsulation—The internal properties of an object, particularly its data, are encapsulated so that they cannot be compromised by the outside world.
- Information hiding—Implementation details are hidden from the user of a class. Ideally, understanding the class interface made up of methods is sufficient to use a class.
- Reuse—The previous points ease the reuse of existing classes. Moreover, inheritance allows to extend and adapt a class for a new purpose while reusing the inherited features. Inheritance can be understood as a grey box reuse technique providing controlled modifications as detailed in [123]. It avoids the error-prone white-box, copy-and-paste style of reuse while not being as restrictive and inflexible as unmodified black-box reuse [117].
- Extensibility—Polymorphism and dynamic binding allow to introduce objects of new classes into a system without having to modify existing code.
- Maintainability—The structuring and standardised documentation provided by an OOA/D model eases the maintenance of complex systems.
- Larger scale concepts—Object-oriented features are the basis for the larger scale software reuse concepts of patterns [57], components [154], and frameworks [52]. Their adaptation to hardware is beyond the scope of this work, but may be of interest for future research.

For hardware modelling and simulation, the supposition of increased productivity [118] has been confirmed, although still on a small statistical basis, by reports that the use of an object-oriented VHDL enables code size reductions of an average 30% compared to plain VHDL [136]. Notably, since these were initial modelling experiments, this has been achieved *without* reuse. An even larger benefit can be expected once reuse libraries become available.

2.5.2 Object-oriented design of embedded systems

Several approaches employ object-orientation in the design of embedded systems. These can be classified into the real-time and co-design categories. Real-time OO techniques such as ROOM [146], Real-Time UML [44], or the object-based HRT-HOOD [68], allow to deal with soft and/or hard real-time constraints in the design of embedded software. More relevant to this work are the following co-design methodologies, which include hardware design.

Model-based object-oriented systems engineering (MOOSE) [109] facilitates an object-based system specification. At first, a target-independent, so-called executable model is specified. Objects of this model are later commit-

ted to an implementation as software, hardware, or firmware. From the resulting committed model, implementation activities start. A VHDL netlist describing the interconnection of hardware objects can be created, and objects can be mapped to existing hardware components. However, the implementation of application specific hardware is not supported. In this case, only empty VHDL templates are generated; the functionality has to be filled in by the user.

In the INSYDE project [67], Rumbaugh's Object Modeling Technique (OMT) [137] has been utilised for system specification. OMT, being fully object-oriented, includes support for inheritance and polymorphism. VHDL code generation techniques for hardware objects and their interconnections have been suggested, including coverage of the objects' functionality. The resulting code can be simulated, but it is not synthesizable. It would have to be refined manually to respect the VHDL synthesis subset [75]. Furthermore, we question that OOD models will be used in practice for a full specification of functionality. Rather, a user will wish to add functional details using the implementation language. This is difficult with the INSYDE approach since VHDL lacks object-oriented features, making it impossible to maintain the original model structure that employs inheritance and polymorphism.

To overcome the deficiencies of the purely graphical approaches MOOSE and INSYDE, we suggest the use of an object-oriented implementation languages. Respective approaches are investigated in the next chapter. In the remainder of this chapter, the use of such languages is assumed.

2.5.3 Object-orientation and synthesis

Once an OO model has been implemented in an OOP language or OO-HDL, it is desirable to automate further processing as much as possible by tools. In the software domain, this is achieved by compilation. For the hardware side, synthesis technology must be developed. Otherwise, the user would have to translate all object-oriented constructs manually into a non-object-oriented form that can be expressed in a standard HDL and processed with current tools. This would be a tedious and error prone task, which has to be repeated each time a modification or correction is made in the OO model.

However, synthesis from object-oriented models is not only a "necessary evil" that follows due to the choice of object-oriented systems engineering. Rather, the user can benefit from an object-oriented modelling style in several ways, compared to modelling for HLS of algorithmic descriptions:

- The user does not have to deal with the synchronisation of processes. These are synthesized independent of each other in HLS, which imposes the task of describing their synchronisation, e.g. by handshaking, on the user. Our object synthesis approach will generate the necessary communication automatically to implement the higher-level concept of message passing between concurrent units.
- Object-orientation allows to model reusable hardware units that maintain an internal state in a behavioural way. At the algorithmic level, this would require the use of procedures together with external variables as it has been described in [79] since procedures alone cannot maintain data values between invocations. In an object-oriented model, the object stores (and encapsulates) this data beyond the execution time of a method.
- Coding for HLS requires adherence to tool-specific coding styles and templates. Particularly, the reset behaviour and eventual synchronisation with a clock must be described explicitly. In the object-oriented model, we need no clock and reset signals. These are generated in the synthesis step. The initial (reset) state can be defined by default values or special methods, constructors.

2.5.4 Object-orientation in the design flow

In this section, we discuss object-oriented hardware specification and synthesis in the context of the complete design space, represented by both the Y-chart and Ecker cube. We show how the design activity of synthesis from object-oriented models—the subject matter of this thesis—is positioned in the design space, and how it can be integrated with further design steps in an HDL-based design flow.

Object-orientation allows to describe objects, which can be considered as concurrent components, and their communications. While the internals of an object may be modelled in an algorithmic style, the concurrency and communication aspects, as well as the potential integration with software design, let us classify object-oriented hardware specifications to be at the system level. An object-oriented specification in the sense of this work takes a behavioural view of the system. Other approaches allow to specify a structural view. This will be discussed in the next chapter.

The classification of object-oriented hardware modelling approaches in the Ecker cube is addressed in [25]. We emphasize the potential of object-orientation to advance the abstraction of data values.

Synthesis has been identified as a design activity that takes a behavioural specification at some abstraction level and creates a structural model whose components are one abstraction level lower. In this sense, synthesis of an object-oriented model (behavioural view at the system level) would create a structural description with components modelled at the algorithmic level. This is exactly the approach taken in this work. We will show how an object-oriented input description can be processed by a tool, the *object synthesizer*, that generates a netlist of components whose functionality is described by algorithmic VHDL code. This enables the integration into an existing VHDL based design flow for synthesizing the design to the level of abstraction required for production. This hardware flow is shown in the right half of figure 1. The further synthesis steps are high level synthesis (HLS) followed by RTL synthesis, logic synthesis and technology mapping, and finally placement and routing.

As we have motivated object-oriented hardware design with the application of object-orientation to hardware-software systems, we shall now complete the picture towards system design. It starts with an OOA/D model that is implemented partly in hardware (HW) and partly in software (SW). The partitioning decision may be manual or automated. Subsequent SW and HW design activities create an OO program and an OO synthesis model, respectively, including HW/SW interface parts. There may be tool support for code generation from the OOA/D model. While the OO synthesis model serves as input for synthesis as to be described, the OO program can be compiled for execution on a development system, on a virtual machine, or cross-compiled for an implementation platform. Co-simulation helps to validate the interplay of software and hardware before the final system integration.

2.6 Summary

Systems design can benefit from object-orientation through its features supporting abstraction, reuse, and extensibility. However, for a consistent and uninterrupted design flow, the capability of synthesizing digital circuits from an object-oriented description still has to be developed.

The direct synthesis from graphical OOA/D models has turned out impractical. It is desirable to apply an implementation language to specify object-oriented models in the detail necessary for synthesis. Respective approaches exist, and the next chapter provides an overview of this related

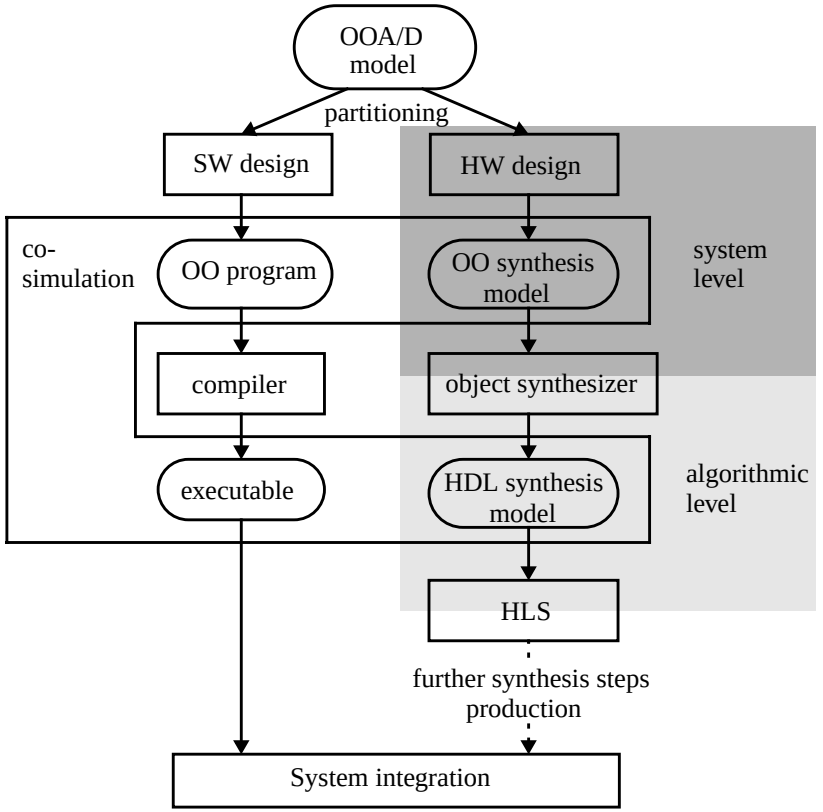


Fig. 1: Object-oriented system design flow

work on language-based, object-oriented hardware description, putting emphasis on synthesis-related aspects.

Likewise, the remainder of this work will be focused on the synthesis aspect. We will develop an approach for the optimized synthesis of an algorithmic level model from an object-oriented, behavioural specification at the system level. This approach allows to delegate the synthesis of sequential algorithms, including the exploration of register transfer level implementations, to a high-level synthesis tool and enables us to interface with established hardware design flows.

Chapter 3

Related Work

The related work on hardware description with object-oriented languages can be classified according to the kind of language used and the style of language usage. The most prominent hardware description languages, VHDL and Verilog, are not object-oriented, but proposals for respective extensions exist. These are distinguished by their preference of a structural or behavioural modelling style [13][110]. Both views are addressed in the next section.

Other approaches suggest the use of a programming language for system specification, including hardware. Some of these employ object-orientation to provide a class library that extends the programming language for concepts necessary or at least useful for system description. Such system description frameworks are presented in the second section.

An alternative is to synthesize programming language based, behavioural object-oriented specifications directly into hardware. This requires a mapping of the object-oriented constructs into an implementation as a digital circuit. The work on this domain is described in a separate, third section.

3.1 Object-oriented HDLs

Many proposals on extending hardware description languages, in particular VHDL, for object-orientation have been published. Only a few, among them SUAVE [10], Objective VHDL [121], and Schumacher's OO-VHDL [142] continue to be actively pursued. All of these provide object-oriented data types which are added to VHDL's type system. Other approaches augment the VHDL design entity with object-oriented features. These are either simple structural inheritance of ports, generics, and processes, or inheritance and communication mechanisms based on more sophisticated interfaces. A com-

ponent-like class construct can as well be found in V++, an object-based language with syntax related to Verilog and C++.

These approaches to object-oriented HDLs are presented in the subsequent sections. Discussion shall be brief and focused on the synthesis aspect relevant to this work. The interested reader can find more detailed reviews of language aspects in [12][17][19][145].

3.1.1 V++

V++ [36] is a synchronous hardware description language based on a class construct with methods and data elements similar to C++. The language has a Verilog-like syntax of statements. Channels can be declared to connect objects, allowing them to communicate by exchange of messages which do, however, not directly correspond to methods. The language can be classified as object-based; since it provides neither inheritance nor polymorphism, it is not a full-featured object-oriented language.

With respect to "hardware semantics," the interesting point about V++ is its mapping to a formal model of computation, a network of concurrent communicating finite state machines (FSMs). Each object can be understood as an FSM. All FSMs are running in parallel and communicate as defined by the message exchange in the object-oriented model. In their paper [36], the authors do not detail the mapping from V++ to the formal model. It is therefore not clear how a method's functionality, specified as an algorithm, is translated into the FSM model. The details of communication synthesis are not revealed either.

3.1.2 Structure based inheritance for VHDL

We classify an OO-VHDL dialect as *structure based* if it augments VHDL's structure modelling constructs, entities and architectures, with inheritance features while not introducing concepts such as methods and polymorphism. The approaches of Ramesh [133] and Ecker [45], as well as some aspects of Objective VHDL [128], belong to this category and are discussed in the following.

Ramesh [133] proposes a syntax for deriving a new entity from an existing one. This is called single inheritance since no more than a single ancestor is allowed. An architecture of such a derived entity inherits from an architecture of its entity's ancestor. If this entity has several architectures, a problem arises since it is not clear nor can be specified from which one to inherit. A

derived entity can add generics and ports, and its architectures may specify additional subprograms with respect to the parent. It is not clear whether other declarations and concurrent statements are inherited and what happens in the case of identifier redefinition. Furthermore there is no indication that in this OO-VHDL dialect entity subprograms can be invoked from outside of the entity. Finally, only VHDL's subprogram overloading exists as a statically resolved ad-hoc form of polymorphism.

Ecker [45] proposes a declaration of a tagged entity from which new entities can be derived. A similar mechanism exists for architectures. By explicit definition of a parent architecture, the case of multiple possible parents is disambiguiated. Since more than one parent can be stated, multiple inheritance is permitted. However, the proposal does not mention how to deal with the arising problems of name conflicts and multiple inheritance of the same feature via reconverging inheritance paths. Features inherited include all available constructs: generics, ports, any kind of declarations, and concurrent statements. The redefinition of an inherited name in a derived unit is said to overwrite the inherited item. The exact semantics of overwriting are not elaborated.

Objective VHDL [128], following Ecker's approach in the structural domain, permits the derivation of new entities and architectures from existing ones, but is constrained to single inheritance. There is inheritance of generics, ports, declarations, and concurrent statements; the latter two can be redefined in a derived unit. The redefinition of a name hides the inherited definition of the same name, but does not replace it in the inherited code. Thereby, the introduction of errors into inherited functionality is prevented.

By a flattening step that involves copying the content of a parent unit into the derived unit and resolving any name conflicts, a model with structural inheritance can be translated into plain VHDL code. Provided that the contents of all entities and architectures, derived or non-derived, adhere to a synthesizable style, the translation result can be processed by a VHDL synthesis tool. This has been demonstrated in [124] for Objective VHDL. Thereby, the structure based approach indeed allows to obtain synthesizable code in a rather trivial way. However, due to the lack of methods that can be invoked from outside an entity and for the missing concepts of polymorphism and dynamic binding, VHDL dialects with structural inheritance do not support a full object-oriented modelling style.

3.1.3 Entity-object based OO-VHDL dialects

To overcome the limitations of structural inheritance, several approaches add the missing features expected from an object to the VHDL design entity. We call these approaches *entity-object based* after the keyword **entity_object** that is used in the Vista OO-VHDL dialect [37] to distinguish the enhanced entity from the traditional VHDL construct. Note that from the object-oriented point of view an entity should rather be named a class while a component instance is similar to an object. Another VHDL extension with entity objects is the LaMI approach [24]. Finally, entity-object capabilities are being developed for Objective VHDL, too. All these approaches have in common entity and architecture inheritance features that are similar to those of Objective VHDL described in the previous section. The major additions and differences are pointed out in the following.

Vista Technologies' entity-object [37] has an interface made up of not only ports, but also so-called operations. The execution of an operation can be requested from outside of an object by sending a corresponding message and addressing the target object by means of an entity handle that is created upon entity-object instantiation. Entity handles are not typed, can be exchanged in a system via signals, and can be stored in variables. It is possible to send any message to a statically unknown entity-object designated by an entity handle. Thereby, polymorphism is introduced. Dynamic binding of a message to an operation takes place at the target object. If no operation corresponds to a message received, a run-time error is raised.

Since the entity-based description style involves concurrency, it is possible that several messages arrive at the same time at a target object, or that new messages arrive while an operation is under execution. In this case, the messages are buffered in a queue for later execution. There is an implicit default mechanism that lets objects accept new messages from its queue automatically, controlled by their arrival order and optional priorities. Alternatively, the user may implement an explicit process that accepts messages using an Ada-like select statement and invokes the corresponding operation. While this allows to model the conditional acceptance of service requests depending on the object's state, the modeling style incurs the risk of running into inheritance anomalies as discussed extensively in [145].

The LaMI proposal [24] adds operation declarations to the features of the regular entity, i.e., there is no special entity-object construct. Operations are similar to VHDL processes rather than subprograms, being able to execute concurrently in a single object. Compared to Vista, this makes the queuing of

different concurrent service requests unnecessary. Still, requests of the same method have to be queued. Concurrent accesses of operations to the object's data are an inherent problem of the concurrent operations. This problem has not been addressed by the language designers.

Entity instances (objects) are addressed by the label of their component instantiation statement instead of an entity handle. Such a label is invisible and cannot be made visible outside of the declarative region of the instantiation. Furthermore, it cannot be stored in variables nor transported via signals. Hence, communication by message exchange can only take place from a given level in the design hierarchy to the objects that are instantiated at this level. This is a severe restriction compared to the communication capabilities provided by object-oriented modelling in general. A further implication is that polymorphism and dynamic binding cannot be enabled; only a broadcasting feature that addresses all objects (instances) of an entity-class is provided as a replacement.

Objective VHDL enables entity-object based descriptions by suggesting a modelling style for the external invocation of subprograms declared in an entity [119][120]. The modelling style is based on the use of signals to transport service requests to a target entity. This approach leaves all communication details, e.g. the communication protocol and the resolution of concurrent service requests, to be implemented by the user. A tool has been developed to automate some recurring routine tasks [51], but modelling is still cumbersome. While there is work towards language support for constructs that enable easier message exchange, respective publications are not yet available at the time of the submission of this thesis. We hope that these constructs will enable the user to deal with synchronisation and conditional request acceptance without running into inheritance anomaly issues.

The existing entity-object based approaches impose significant problems on a synthesis tool. Entity handles are even worse than pointers in VHDL (variables of access types) in that they are untyped. Pointer synthesis is known to be a difficult problem that involves worst-case analyses of the set of potentially addressed objects [147]. Any object included in this set but not actually addressed during system operation incurs a hardware overhead. VHDL synthesis tools do not synthesize models that use pointers for this reason. Similarly, we should avoid entity handles in object-oriented synthesis models.

Message queues are another major issue. While it is possible to implement queues in hardware at a non-neglectable cost, a synthesis tool would have to know the required number of entries in a queue. If this number is pro-

vided by the user or determined by non-exhaustive simulations, it may turn out too small during actual system operation. A worst-case analysis, if feasible at all, would be on the safe side but lead to hardware overhead.

3.1.4 Type based OO-VHDL dialects with class constructs

Class type based approaches to object-oriented VHDL are characterised by defining classes as data types (instead of entities) and objects as VHDL data objects such as variables or signals (as opposed to component instances). The following approaches can be summarised under this category and are discussed in the remainder of this section: VHDL++ [58], VHDL_OBJ [171], inheritance of protected types [164], Cabanis' approach [33], and Salamunic-car's approach [139].

VHDL++ [58] adopts the class construct of C++ into VHDL. However, it relies on a selected name mechanism rather than pointers to address objects for inter-object communication. The language does not provide polymorphism. For communication between concurrent objects, classes are augmented with ports that allow the user to implement a protocol based on signals. The resolution of concurrent requests has to be modelled in the object body (implementation). This is prone to inheritance anomaly issues. Thanks to its construction, VHDL++ can be considered synthesizable; a translation into VHDL has been suggested for large parts of the language. However, due to the lack of support for message passing between concurrent objects and polymorphism, the modelling power does not suffice for a hardware implementation of general object-oriented models.

VHDL_OBJ [171] provides a similar C++ based class construct. Different to VHDL++, methods can be declared as virtual to enable dynamic binding together with the use of pointers, which allows and is required to employ polymorphism. Hence, synthesis is infeasible at least when this important modelling feature is used. The instantiation of objects is limited to processes, making the object-oriented features usable in a sequential context only. This obviates the need to deal with concurreny issues, but also limits the value of VHDL_OBJ for modelling concurrent hardware.

Protected types [165] are a proposed concept to control concurrent access to shared variables, which have been introduced in VHDL'93 [74], by means of a monitor concept [66]. Protected types are similar to a class in that they encapsulate their instances' internal state, allowing access only via operations declared in the type's interface. The addition of inheritance [164] targets at making protected shared variables a means for object-oriented

modelling. However, while the monitor concept provides mutual exclusion semantics, it does not cover the need for condition synchronisation. Moreover, dynamic polymorphism is not provided. Since shared variables are not included in the synthesizable VHDL subset [75], we must rule out this approach for hardware synthesis.

Cabanis' classification-oriented proposal [33][34] comes with a class construct that can be used for declaring signals. Inheritance is provided, and methods can be called similar to C++. Polymorphism and dynamic binding are not available. With respect to concurrency, the language lacks a definition of a mechanism that applies when multiple concurrent processes invoke methods of the same object. The user has to model the resolution of such concurrent accesses on his own. Hence, while synthesis is possible, we must conclude that only limited support for concurrent object-oriented hardware modelling is provided.

Salamuniccar [139] has proposed another class based approach with concurrent objects. Its distinguishing feature is the possibility to define processes inside the class type. A process could be used like an object body in COOP to model explicit acceptance of messages and condition synchronisation. However, this incurs the problem of inheritance anomalies. Multiple processes could even introduce concurrency inside an object, but the user would be responsible to take care of avoiding resource conflicts when, e.g., attributes are accessed. While there is no work on synthesis or a translation into VHDL, the class features can be considered as synthesizable. Polymorphism, however, would have to be excluded from synthesis because it relies on pointers.

3.1.5 Type based approaches with Ada 95 background

Since VHDL's syntax and partly its semantics have been designed following the example of Ada [60], it is obvious to consider the object-oriented extensions introduced in Ada 95 [22] for OO-VHDL. Respective approaches have been undertaken by Mills [108], Ashenden at al. [10], and Schumacher [142].

Mills' proposal [108] focuses on the addition of Ada's syntactic constructs for object-oriented modelling to VHDL's grammar while not addressing their semantic integration. New features include extensible tagged types for inheritance, private types and private parts in packages for encapsulation, the declaration of subprograms as primitive operations (methods) of tagged types, and a class-wide type for polymorphism. The feasibility of hardware synthesis is limited since access types and pointer variables may be

employed in object-oriented models. This is necessary in particular when polymorphism and dynamic binding are to be used.

The focus of SUAVE [10][11][14][16][18], with its additional constructs for communication modelling and dynamic process creation [15], is clearly on system-level modelling. Work on synthesis or a translation into VHDL has not been published. Still, the object-oriented mechanisms of the language can largely be considered as synthesizable. Their syntax is similar to Mills' approach; however, semantics is specified in detail and with some deviations from Ada so as to be more suitable for hardware modelling. Particularly, SUAVE permits the declaration of signals of an object-oriented type (tagged record or class-wide type). This facilitates the use of object-orientation in a concurrent context and allows to use polymorphism without pointers. Polymorphic variables, however, still have to be declared with an access type.

While SUAVE developers target the implementation of native tools for their language, Schumacher describes for his Ada-based OO-VHDL dialect a translation of the object-oriented extensions into plain VHDL [142][145]. This allows to use VHDL tools for further processing. The language itself is similar to SUAVE, but advances synthesizability by avoiding access types and the use of pointers completely. This involves a value-based semantics of polymorphic variables, as opposed to pointers. It is claimed that the VHDL constructs created during translation are synthesizable. Hence, a complete object-oriented specification is synthesizable if it employs the object-oriented extensions together with the synthesizable VHDL subset.

However, the generated VHDL is not optimised for synthesis. The translation of class-wide types can lead to a large register or memory overhead because no bit-optimal encoding is used. Moreover, the limitations of VHDL synthesis tools are not considered. Translation of primitive operations (methods) makes use of subprograms which are effectively inlined by synthesizers. Therefore, a method is synthesized separately for every single call. It is not feasible to synthesize it once and let the synthesis tool instantiate the result to implement the method invocations, possibly mapping several invocations to a common resource. Likewise, an object cannot be mapped to a previously synthesized or designed library element. This holds even if vendor-specific library mechanisms such as Synopsys' Designware [153] are used since these usually cannot deal with operations that have unbounded delay or bidirectional (INOUT) parameters, which is typically the case for methods.

Another problem is that all operations that modify the state of a particular object must be invoked from a single sequential process; otherwise, a non-synthesizable resolution function would have to be introduced. Any concur-

rent use requires the user to explicitly model inter-process communication using signals. Schumacher describes a modelling style which allows to implement this communication with an innovative approach for condition synchronisation [144][145]. However, the modelling effort is large and the modelling style is not synthesizable.

Finally, it is desirable to have a more detailed idea of (and control over) the circuit structure resulting from synthesis in order to make the whole process more transparent and predictable to designers. This requires the definition of an at least informal “hardware semantics“ for object-oriented constructs.

3.1.6 The type aspect of Objective VHDL

Objective VHDL [122][126][129][130][131], while providing additional entity-based inheritance as described before, is another VHDL dialect with type-based object-orientation. The language employs a class construct like in C++ instead of extensible records with separate operations. This choice has been made for the convenience of programmers, most of which are more familiar with C++ than Ada. Still, the intentions and mechanisms behind the syntax are similar to those of Schumacher’s Ada-based approach.

Hence, Objective VHDL supports the full range of object-oriented concepts: classes with data attributes and methods, inheritance of classes, polymorphism, message passing, and dynamic binding. Furthermore, concurrent modelling is facilitated by supporting class-typed and polymorphic signals. Following Schumacher’s approach, the concept of mutually exclusive guarded methods can be provided for arbitration of concurrent requests and condition synchronisation without inheritance anomaly issues.

Objective VHDL is employed in this work to implement object-oriented models in a hardware description language based form and to demonstrate the synthesis of such descriptions. The language and its application to concurrent object-oriented hardware modelling will be presented in more detail in chapter 5.

3.2 System description frameworks

Hardware description languages have been designed to incorporate some concepts that are not found in most programming languages: simulated time, event-driven simulation, hardware-efficient data types, structure, concurrency, reactivity, and basic mechanisms for communication between concurrent domains. Many of the new synthesis tools for C/C++ are therefore

restricted to the isolated synthesis of sequential algorithms, leaving concurrency and communication out of consideration. The tool C2VHDL from CLevelDesign (formerly Compilogic) belongs to this category. The same holds for Frontier Design's ArtLibrary/ArtBuilder, which provides special support for fixed-point numbers.

There are, however, approaches to including concurrency and communication aspects by either extending the programming language or providing a library that adds the missing functionality. In some sense, this means augmenting the programming language with features that are readily available with HDLs. Yet, advantages can be seen in possibly better execution performance of programming languages and in easier connection of embedded software and hardware development in a homogeneous framework.

In the following, we discuss these approaches according to the languages used: early work using early object-oriented languages, approaches based on C++, and Java based approaches. Another category describes approaches that rely on knowledge and data bases rather than programming languages.

3.2.1 Early work

Takeuchi [155] was among the first to present the idea of object-oriented CAD environments for hardware description. The basic concept is to provide an object-oriented library of concepts common in hardware engineering, such as state machines, netlists, and truth tables. These can be instantiated and personalised by the CAD user. Furthermore, connections can be described. The resulting structural description can be used as input for further analysis, simulation, and synthesis.

Another early work was presented by Pawlak in [116], showing how the object-oriented features of the programming language LOGLAN could be used to model the instantiation and interconnection of gate-level primitives and to execute a simulation. In addition, modelling of the layout was addressed. However, this work does not cover the high-level modelling and synthesis aspects relevant today and to be targeted by this work.

A slightly different approach has been proposed by Wolf [162]. The building blocks of the so-called Fred environment are standard hardware devices rather than standard concepts. They include, e.g., devices of the TI 7400 series. Classification is utilised to group the objects into families (e.g., TTL) and functional categories (e.g., primitive gate, ALU, or register). The Flavors language, an object-oriented dialect of Lisp, is used for describing these relationships.

In his dissertation [140], Sarkar describes an object-oriented design framework named DOORS. Hardware components are modelled as objects using OHDL, an object-oriented HDL developed by the author. From these descriptions, data flow and control flow are extracted into intermediate forms called object dependency graph (ODG) and message flow diagram (MFD), respectively. A controller and datapath can be synthesized from these design representations using high level synthesis techniques. Inheritance plays a role to provide a pre-defined class hierarchy of design primitives. Primitive data types such as bit and integer are polymorphic. However, nowhere the derivation of a new hardware object with additional or redefined methods from an existing one is described. Likewise, polymorphism and dynamic binding are not used with hardware objects. In consequence, no synthesis concepts are presented for these object-oriented features.

3.2.2 Object-oriented modelling of design data

More recently, Wolf's approach has been taken further by Agsteiner and Monjau [1][2]. They describe a hierarchical object-oriented model of hardware devices. A knowledge base and constraint nets are utilised to store the information about classification and hierarchy relationships.

Object-oriented classification and data base management of hardware devices have been applied to build reuse repositories. In [170], a system for the the management of electronics design and production data is described. Barna and Rosenstiel [20][21] focus on the management and retrieval of intellectual property blocks in their Reuse Management System. The language Objective VHDL has been used as well as plain VHDL to describe interface information that allows to automatically assess the suitability of an IP block for a given purpose by means of similarity metrics.

These approaches, while descring important characteristics with the help of object-orientation, do not specify the *functionality* in an object-oriented way. They are therefore out of scope for our further considerations on synthesis from object-oriented models.

3.2.3 C++ based approaches

Recent approaches apply techniques similar to section 3.2.1 at a higher level of abstraction, targeting HW/SW co-design, which includes the hardware synthesis aspect covered by this thesis. Among the first in this direction has been Scenic [99]. This environment is based on C++ and a programming

library which uses the object-oriented features to provide concurrency, reactivity, hardware-oriented data types, and the means to describe the interconnection of components by port mapping. All static structure must be modelled in the constructors, and a dedicated entry function describing dynamic behavior must be provided. Hardware synthesis is possible by elaborating the constructors and implementing the entry function using register transfer or high level synthesis.

Ocapi from IMEC [141] is another object-oriented programming environment based on C++. It facilitates hardware modelling with signal flow graphs and finite state machines and provides a path to synthesis. However, the models being synthesized are flow graphs and state machines but no objects in the object-oriented sense. The focus of CoWare [100][158][159] is on architecture mapping and evaluation of system descriptions in C/C++. Its hardware synthesis capabilities are limited to a subset of C called RT-C that is clearly not object-oriented.

We shall also mention CynApps' C++ based environment for hardware description [38], which primarily targets event-driven simulation and RTL design. Similar to Scenic and Ocapi, this library employs object-orientation to provide an environment for hardware modelling. The objects of a description that uses any of these libraries serve to extend C++ so as to describe concepts such as netlists and concurrency, but do not advance object-oriented hardware specification.

PAM-Blox [105] is a C++ based object-oriented circuit generator for Xilinx FPGAs. It allows to describe parameterisable modules of hardware objects such as counters, shifters, or adders, as well as their instantiation and interconnection. A path to synthesis is provided by the tool PamDC, which translates the structural C++ description into a Xilinx netlist file (XNF format). The hardware objects are mapped to corresponding Xilinx macros.

The Scenic environment has been continued by Synopsys under the name Scenery. It has recently been brought into the SystemC initiative [23] which is joined by other proponents of C based hardware design such as Coware and Frontier Design. As of November 1999, there is no information available about new technical advances.

3.2.4 Java based system modelling

A couple of approaches exist that apply Java for system level modelling and partly synthesis, namely the Programmable System Architecture (PSA) from

Improv Systems [97], JavaTime [167], Reactive Java [115], and the work of Helaihel and Olukotun [64].

The approach of Improv Systems [97] is based on a system-on-chip multiprocessor architecture that can be programmed via a very long instruction word (VLIW) instruction set [65]. A compilation environment named Solo performs the mapping of a Java program to this architecture. Concrete architectures such as Jazz being defined and chips being supplied by Improv, architecture synthesis itself is out of scope. Only software synthesis, but no hardware synthesis is performed.

JavaTime [167][168][169] is a set of tools supporting the successive, formal refinement of a Java program into a particular model of computation called abstractable synchronous reactive (ASR) model, suitable for the specification and synthesis of reactive embedded systems. Since Java's capabilities go beyond what can be expressed as ASR, a policy of use must be imposed on the source language. The policy forbids, for instance, the dynamic instantiation of objects and demands that all objects be instantiated during an initialisation phase. This can be considered a reasonable assumption reflecting the static nature of hardware. However, object-orientation in Java is based on dynamic mechanisms such as references (restricted-use pointers) and virtual methods. It is therefore unlikely that the policy of use can be formulated so that true object-oriented behaviour can be synthesized. Instead, an example in [168] suggests that all behaviour of an object must be specified in a dedicated method. Hence, the object itself is not synthesized. It rather provides a framework for specification, similar to the Scenic approach, but defined by a policy of use instead of a programming library.

With Reactive Java [115], a slightly different approach is taken. The authors describe a class library that provides control structures for concurrency, sequencing and preemption that are closely related to those of the synchronous language Esterel. Their use makes the translation into an abstract, synchronous reactive representation easier compared to JavaTime. Again, classes and objects provide a framework of hardware specification concepts, but are themselves not synthesized into hardware.

Helaihel and Olukotun [64] describe the use of Java for algorithmic specification of HW/SW systems. Java bytecode is analysed to resolve memory references, dynamic memory allocation, and the dynamic linking of Java programs. This allows to generate a control/dataflow graph which can be used as input to hardware synthesis. The process of generating this graph and the successive synthesis step are not detailed any further.

3.3 Synthesis of OOP language features

There are several approaches which go beyond the programming frameworks presented in the previous chapter in that they target to synthesize directly from an object-oriented language. Thus, they have the potential to synthesize hardware from a specification that employs full object-orientation for modelling behaviour. In the following, we will investigate how this goal is achieved with JavaSynth [87], Matisse [86], and SystemC++ [104].

3.3.1 JavaSynth

In [87][88][89][90][91], Kuhn et al. describe the use of Java for hardware modelling. While the emphasis of the publications is on simulation, there are two interesting points with respect to hardware synthesis.

First, the authors describe a behavioural and a structural interpretation of Java. In the behavioural interpretation, an object is understood as a state variable whose value is transported via wires. A method corresponds to a hardware component that implements the functionality described by the methods. The object state and the method's parameters are both inputs and outputs of the hardware component since they may be read and modified. A return value would become an output. The structural interpretation, on the other hand, views the object itself as a hardware component that receives via wires information about a method to be executed. Hence, the structural interpretation has the potential to implement Java objects directly in hardware. On the other hand, state is separated from functionality in the behavioural interpretation, which reflects the problem presented in the second point of section 2.5.3.

Secondly, the authors mention a tool named javaSynth that applys the structural interpretation to a Java program and creates a corresponding VHDL netlist. The objects are mapped to existing, pre-synthesized VHDL components. Moreover, behavioural VHDL code can be generated that is equivalent to the methods and suitable for processing by VHDL synthesis tools. However, details about the VHDL code generation and synthesis process are not revealed in the published material.

3.3.2 Matisse

Matisse [39][40][41][86][160] targets the synthesis of the dynamic allocation and access of complex memory data structures, with a focus on data-intensive telecommunication applications. It employs C++ class types extended

for concurrency (“active classes“) for the specification of the data structures and their manipulation through methods. The mapping of data to an addressable memory allows to synthesize pointers as values to be put on the address bus. This is close to the implementation of software on a processor and provides the potential to implement inheritance and polymorphism like a software compiler would do. However, in contrast to the typical hardware model of independent registers connected by combinational logic, the memory limits concurrency because the number of values to be accessed at a time is constrained by the number of memory ports. This architecture is justified when data structures are large or must be shared with software, but it is too special to serve as a target for synthesis from general object-oriented descriptions.

3.3.3 SystemC++

SystemC++, described in [104], is based on an extension of C++ for concurrent objects and their communication. Dynamic object creation and deallocation are supported in hardware by a dynamic, hardware-controlled binding of objects to a static set of resources. The hardware target architecture is similar to software in that it implements a kind of run-time system for the execution of an object-oriented model. Thereby, many software concepts can be implemented in hardware.

Different to software, the architecture is not based on a global memory, but on smaller, concurrent units to which objects are mapped. It reminds of an application specific processor with multiple execution units running a hard-wired program. This certainly extends the scope of hardware synthesis. However, it is open whether the overhead for dynamic hardware resource management is acceptable. Moreover, dynamic resource management impairs predictability as the system may run out of resources. To avoid this hazard, coding styles for embedded software development often discourage dynamic resource allocation [32].

3.4 Summary

Various approaches to employing object-orientation in systems or hardware design have been evaluated for their coverage and use of object-oriented concepts as well as their synthesizability. It is important to note that synthesizability refers to the feasibility of synthesis from the object-oriented description, but does not imply actual work on tool support for synthesis. In fact, there are only few reports on such activities.

We have pointed out that augmenting hardware description languages with simple structural inheritance is not powerful enough to support the concepts used in OOA/D models. The support for more advanced concepts such as message queuing and reference exchange between structural entities, on the other hand, impairs the synthesizability of object-oriented models. Type based approaches supporting a behavioural modelling style are more promising with respect to synthesis.

Many programming language based approaches use object-orientation as a means to provide a framework that enables hardware modelling. However, hardware synthesis of the object-oriented constructs themselves is not provided. The modelling style to be applied for hardware is quite different from the way the OOP language would be used in software programming. Hence, these approaches do not provide an integrated approach to co-design. While a single language is used, hardware and software remain separated.

The hardware synthesis of object-oriented programming language constructs, comparable to their compilation into executable software, is addressed by only a few researchers. Some propose special architectures that ease the mapping of dynamic software concepts such as address dereferentiation and object creation. Others rely on analysis techniques to transform dynamic behaviour into static structures.

This work takes a different approach. We suggest an interpretation of object-oriented models (i.e., a meta-model) that facilitates the direct implementation of object-oriented concepts as traditional, application-specific digital circuits. Hardware structures correspond to, e.g., derived objects, polymorphism, and dynamic binding, and are transparent to the designer.

Emphasis will be put on the latest advances in concurrent object-oriented modelling. We investigate the synthesis of condition synchronisation based on guarded methods so as to enable designers to use inheritance and concurrency together while avoiding inheritance anomaly issues.

Our conceptual considerations abstract from a specific implementation language in the formulation of concepts, hoping to overcome language barriers. The later use of Objective VHDL, designed to match the staticness requirements of hardware models, will allow us to present a practical implementation of our approach.

Chapter 4

Object-Orientation: A Hardware Perspective

In the first half of this chapter, we develop a meta-model of object-orientation that is suitable for a digital circuit implementation. In order to achieve a language-independent view on object-orientation, we consider the object-oriented analysis and design notations of the Unified Modeling Language instead of a specific object-oriented programming or hardware description language. The features of an object-oriented model, namely objects and their classification, communication between objects, and polymorphism, are investigated in the three following sections.

In two further sections, hardware implementations are devised for the meta-model's aspects related to objects and their interconnections. Where appropriate, language implementation mechanisms known from software programming languages are adopted. Otherwise, a special hardware interpretation of an object-oriented feature is devised, or a usage restriction pointed out where necessary.

4.1 Objects and classes

An object is characterised by its state, which can take on a value out of the state space defined by its class, and by the services it provides to its environment. We show how these properties are extracted from an object-oriented model and captured in our meta-model. The presentation includes coverage of derivation, associations such as composition and aggregation, and parameterization. A transformation into a finite state machine model shows the feasibility of implementing an object in hardware.

4.1.1 Types and value sets

All high-level languages have a notion of types as an abstraction of values. Typically, a language provides some primitive types and mechanisms for the construction of user-defined types. Primitive types correspond to pre-defined value sets; e.g., an integer type may represent the integer values -2^{31}, $-2^{31}+1$, ..., 0, ..., $2^{31}-1$. A user-defined type can be declared by defining its value set explicitly (e.g., enumeration type), deriving it from an existing type, or by composing types (e.g., as a record or array). In the latter case, the value set is defined by the cartesian product of the types of the elements.

The notion of types and values is captured by the following definitions which will be used in the remainder of this work:

- *Types* is the universal set of all types declared in a system.
- *Values* is the universal set of all possible values. Values may be scalar (integer, real, enumeration value) or composite (record, array value). Non-synthesizable values such as physical values and pointers [75] are excluded from our considerations.
- $t \in Types$ is a meta-model variable that stands for some type. Concrete type identifiers have a capital first letter to distinguish them from type variables of the meta-model.
- $S : Types \rightarrow \wp(Values)$ is a mapping from the set of types into the power set of values. The mapping yields for a type, $t \in Types$, its corresponding value set, $S(t) \subseteq Values$.

Note that two types can share some or all of their values. The type is not identified with its value set. It depends on the specific language whether two different types with identical value sets are compatible. Likewise, a language may allow the definition of compatible subtypes with narrowed value sets. For the following, language-independent considerations, only the value set is relevant, not the compatibility information. A subtype is therefore handled like a type of its own.

Every data object such as a constant or a variable is declared with a type and can take on any value from the type's value set. The value set will also be called the *state space* of the data object. In the following, we will extend this notion to object-oriented data types.

4.1.2 Object state space

Each object in an object-oriented model has an individual state which may vary over time. The state space is defined by a class declaration which can be understood as a template for the creation of similar objects. Figure 2 shows

the UML notation of a class type, $C \in Types$, which we assume to be a non-derived class. The class specifies the state space of each of its objects as a collection of so-called attributes. This term, to be told apart from VHDL attributes, means the definition of a variable data field with a given type.

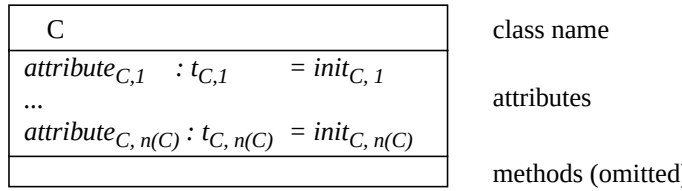


Fig. 2: Attributes defined in a UML class diagram

Be $n(C) \in \mathbb{N}$ the number of attributes defined in class *C*. For the value of each attribute holds: $attribute_{C,i} \in S(t_{C,i}), i \in \{1, \ldots, n(C)\}$. The state space defined by the class, *S(C)*, follows as the cartesian product of the attributes' value sets:

(1) $S(C) = S(t_{C,1}) \times \ldots \times S(t_{C,n(C)})$.

An initial value, $init_{C,i} \in S(t_{C,i}), i \in \{1, \ldots, n(C)\}$, can be defined explicitly for each attribute as shown in figure 2. The UML allows to omit the explicit definition. In this case, a default value or random initialisation is assumed according to the mechanisms of the implementation language. The resulting initial state of an object of class *C* is:

(2) $init_C = (init_{C,1}, \ldots, init_{C,n(C)}) \in S(C)$.

Note that accessibility levels such as public, private, and protected may be specified. These affect the use of attributes in the object-oriented model, but not the state space defined, and can therefore be ignored for our purposes.

4.1.3 Methods

By defining methods in a class declaration as shown in figure 3, the services that each object of the class provides to its environment are specified. A method, when invoked, may receive inputs required for the service and create outputs as a result of the service. Moreover, a method can modify an object's state. In the strict sense of object-orientation, this should be the only way to cause a state transition of an object.

A method is similar to a subprogram. Its interface is characterised by the method's name and parameter list. The parameter list can include input, out-

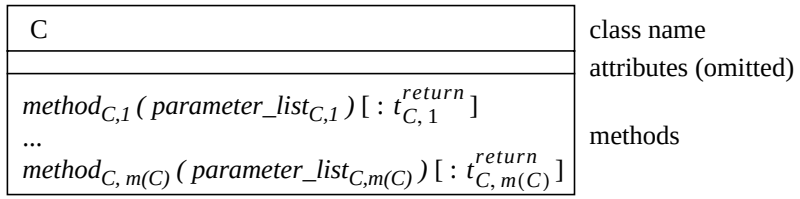


Fig. 3: Method specification in a UML class diagram

put, and bidirectional parameters. In addition, a function-like method has a return value. Each parameter, as well as the return value, is of a specified type. The following considerations will be eased by considering a return value as an output of a method, and by splitting each bidirectional parameter into a separate input and output.

Be $m(C) \in \mathbb{N}$ the number of methods specified for the class. From each method, $method_{C,i}$, $i \in \{1, \ldots, m(C)\}$, parameter definitions are extracted as follows:

- Input and bidirectional parameters, $p^{in}_{C,i,j}$, of type $t^{in}_{C,i,j}$, where $j \in \{1, \ldots, k(method_{C,i})\}$ and $k(method_{C,i}) \in \mathbb{N}$ is the number of these parameters. The type specifies a parameter's state space, $S(t^{in}_{C,i,j})$. The overall input space, $S(in_{C,i})$, of $method_{C,i}$ follows as the cartesian product of these individual parameter state spaces:

(3) $$S(in_{C,i}) = S(t^{in}_{C,i,1}) \times \ldots \times S(t^{in}_{C,i,k(method_{C,i})}).$$

- Output, bidirectional, and return value parameters, $p^{out}_{C,i,j}$, of type $t^{out}_{C,i,j}$, where $j \in \{1, \ldots, l(method_{C,i})\}$ and $l(method_{C,i}) \in \mathbb{N}$ is the number of these parameters. With parameter state spaces $S(t^{out}_{C,i,j})$, the overall output space, $S(out_{C,i})$, is:

(4) $$S(out_{C,i}) = S(t^{out}_{C,i,1}) \times \ldots \times S(t^{out}_{C,i,l(method_{C,i})}).$$

If a method does not have input or output parameters, the respective space is the empty set. With these notations, a method can be understood as a mapping from the current state of the object and the input parameters to the next state and the output parameters:

(5) $$method_{C,i} : S(C) \times S(in_{C,i}) \rightarrow S(C) \times S(out_{C,i}).$$

Depending on the implementation language, special methods called constructors may be defined to describe an initialisation of objects. Other methods may be pre-defined, for instance equality and assignment operators.

4.1.4 Derivation

Derivation is a mechanism for defining a new class by extending one or several existing classes, the so-called parent classes. The derived class is said to inherit from its parents. We speak of *single inheritance* if only a single parent class is allowed. *Multiple inheritance* from more than a single parent has turned out to be problematic due to, e.g., name conflicts between declarations inherited from different parents. Modern object-oriented languages such as Ada 95 and Java therefore restrict the derivation mechanism to single inheritance. The same assumption is made in this work. Application scenarios for multiple inheritance can be handled well with the help of mechanisms like interface inheritance and genericity which are discussed later in this chapter (see section 4.1.8).

A derived class, $D \in Types$, is defined by its parent class, *P*, and by its own attributes and methods. The parent class can be a non-derived class or a derived class. It is denoted by an arrow from the derived class to the parent class in the graphical UML notation as shown in figure 4. The arrow represents the *is-a* relationship: *D is-a P*. This relationship can be interpreted in a way that the derived class *is a* more specialised variant of the parent.

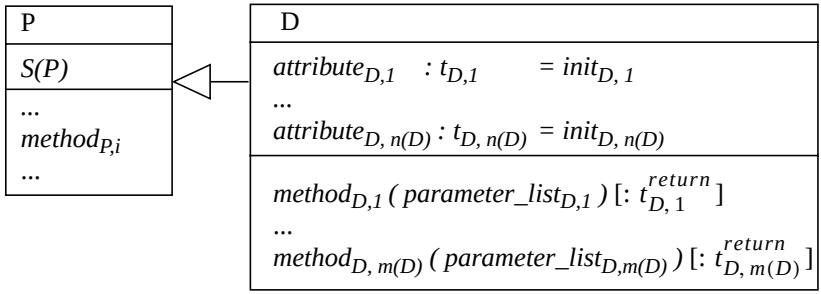


Fig. 4: Derivation in UML

In order to be well-defined, a derived class must not be the parent of one of its direct or indirect ancestors. If this were the case, the class definition would be circular, i.e., depend on itself. Hence, the inheritance relationship must not have cycles. This property, together with single inheritance, ensures that the *is-a* relationship can be represented as a collection of *inheritance trees*. The root of each tree is a non-derived class, the other nodes represent derived classes, and edges represent the parent-child relation.

The new attributes and methods of the derived class are specified in the UML diagram in the same way as described in the previous sections. In addition, the derived class inherits all attributes and methods from its parent. The state space defined by the derived class can therefore be expressed as an extension of the parent's state space:

(6) $S(D) = S(P) \times S(t_{D,1}) \times \ldots \times S(t_{D,n(D)})$, and the initial state as:

(7) $init_D = (init_C, init_{D,1}, \ldots, init_{D,n(D)}) \in S(D)$.

Similarly, the services available from objects of the derived class include not only the methods specified in the class itself, but also those inherited from the parent. This includes methods inherited by the parent if the parent is another derived class. Using the transitive closure, *is-a**, of the *is-a* relationship, the services of *D* can be expressed as the set of all methods declared in any, direct or indirect, parent *p* of *D*, including *D*:

(8) $M_D = \{method_{p,i} \mid D \text{ is-a* } p, i \in \{1, \ldots, m(p)\}\}$.

Note that the inherited methods operate on the state space of the class in which they are defined. Their adaptation to the extended state space of the derived class is covered in the following section.

Finally, it should be mentioned that an inherited service can be re-defined in a derived class by defining a method with the same name and parameter list. Depending on the semantics of the implementation language, the new method may replace the inherited one (as with C++ virtual methods), or both may co-exist (as in Ada 95). In this work, co-existence is assumed. Replacing of re-defined services can still be modelled by delegating all invocations of the service to the new method.

4.1.5 The object as a finite state machine

A finite state machine (FSM) is characterised by its sets of states, inputs, and outputs, next state and output functions, and an initial state. Similar terms have been used in the previous sections to define a mathematical meta-model of classes. We will now show how the structures described by a class can be expressed by an FSM model. FSMs are frequently used for hardware specification, and their implementation is well understood.

The state space, $S(C)$, of a class can be used directly as the set of states of the corresponding FSM. The input spaces of all methods from *C* and all its parents are merged into an overall set of inputs, I_C, of the FSM, and the analogue is done to define the FSM's set of outputs, O_C:

(9) $I_C = \bigcup_{C\ is\text{-}a^*\ p} \bigcup_{i \in \{1, \ldots, m(p)\}} S(in_{p,i})$; O_C analogous.

In order to match the FSM model, each method, $method_{p,i} \in M_C$, is split into separate next state and output functions, $next_{p,i} : S(p) \times I_C \to S(p)$ and $outp_{p,i} : S(p) \times I_C \to O_C$. These should compute the same state transitions and outputs as the original method for all states, $q_p \in S(p)$, and all inputs, $r \in S(in_{p,i})$:

(10) $method(q_p, r) = (next(q_p, r), outp(q_p, r))$.

The *next* and *outp* functions can be obtained by projection of the method's co-domain (range) to $S(p)$ and O_C, respectively.

Note that the domains and co-domains of the separate next state and output functions have been adjusted to the FSM input and output sets, I_C and O_C. These are supersets of the original method's input and output spaces. This is reflected in that the *outp* function uses only part of the output space and in that the function values of *next* and *outp* are explicitly defined only for a subset, $S(in_{p,i}) \subseteq I_C$, of their input domain. The other function values may be chosen randomly.

Two things remain to be done: First, all inherited methods have to be adapted to the state space of the class and its equivalent FSM. Secondly, the multiple next state functions and output functions must be combined into a single state transition function and output function of the FSM, respectively. Both goals are achieved by defining $next : S(C) \times (M_C \times I_C) \to S(C)$ and $outp : S(C) \times (M_C \times I_C) \to O_C$ so that for all $method_{p,i} \in M_C$, for all $r \in S(in_{p,i})$, and for all $q_C = (q_p, q_{ext}) \in S(C)$ with $q_p \in S(p)$ holds:

(11) $next(q_C, (method_{p,i}, r)) = (next_{p,i}(q_p, r), q_{ext})$ and

(12) $outp(q_C, (method_{p,i}, r)) = outp_{p,i}(q_p, r)$.

These functions receive a service request via the $method_{p,i}$ parameter and compute the next state and output according to the individual method definition which corresponds to the request. The *next* function affects only that portion, q_p, of the state which belongs to the class, p, in which a method has been defined. The other attributes, q_{ext}, are left unmodified.

Now an object of class C can be implemented by the finite state machine

(13) $FSM_C = (S(C), (M_C \times I_C), O_C, next, outp, init_C)$.

This FSM should be understood as a conceptual model. We will not target direct synthesis from, e.g., its state table. In many practical cases, the state space will be too large to apply typical FSM synthesis algorithms such as state minimisation and encoding optimization. Likewise, methods may dis-

play complex, possibly data-dependent behaviour which cannot be computed in a single clock cycle. Hence, the state transition and output functions may require a sequential implementation. These issues are addressed in section 4.4.

4.1.6 Object life-cycle

Objects are created by instantiating classes. This action is also called object allocation. It is dynamic, i.e. performed at run-time, in most programming languages. For example, Ada 95, C++ and Java provide the **new** operator to create an object. Storage for the object's state is allocated in heap memory. A special method, the constructor, is invoked by the run-time system in order to initialise the object. The constructor may be implicitly defined or provided by the user.

Similarly, objects can be destroyed while the system is at operation. A destructor method, defined explicitly or by default rules, describes how the object is finalised. This includes returning its state memory to the pool of free resources maintained by the run-time system. The destructor may be invoked explicitly by the user as in C++. An alternative is garbage collection as found in Java, where a run-time routine occasionally inspects whether objects are still being referenced and performs finalisation if this is not the case

In any real-world system, memory resources are limited. Object allocation fails if the number of objects created exceeds the available resources. Moreover, it is difficult to assess the maximum resources needed if objects are dynamically allocated. In many coding styles, especially for programming of safety-critical systems, dynamic allocation is therefore restricted [32]. Good practice includes to allocate all resources at system start-up so that no failure can occur during normal operation.

Considering object-orientation for hardware, the static nature of hardware discourages dynamic allocation even further. The dynamic allocation of hardware resources would be possible only with a special circuit technology, dynamically reconfigurable field programmable gate arrays (FPGAs). However, this work shall not be restricted to these devices. An alternative is the static allocation of a pool of resources to which objects could be dynamically bound as in the SystemC++ approach (cf. section 3.3.3). While being technology-independent, this approach still suffers from the inherent resource limitations which may cause object binding to fail or to stall until an another object is finalised. Both effects impair the predictability of system behaviour.

Moreover, non-trivial run-time resource management would have to be implemented in hardware.

Therefore, this work's meta-model of object-orientation only provides static allocation and binding of objects. It is assumed that objects created by instantiating a class can be determined by some static analysis. Furthermore, objects shall exist during the complete life-time of the system, i.e., there is no finalisation. In the following, the class C_i will be understood as the set of its objects, $obj \in C_i$. The static set of all objects, Ω, follows as the disjoint unification of all classes:

$$\Omega = \dot{\bigcup_i} C_i \quad . \tag{14}$$

4.1.7 Associations

Having considered individual objects up to now, we shall investigate their relationships in this section. In the UML, these relationships are described as associations between the objects' classes. An association is characterised by its name, the classes involved, and the roles and multiplicities of objects of these classes:

- The association name indicates the meaning of the association.
- The classes constrain the objects involved in the association. An association may include objects of two classes (binary association), three classes (ternary association), or more.
- Objects of these classes have certain roles in the relationship.
- Multiplicities define how many objects of each class take part in an association. The UML provides the notations '1' (exactly one), '0..1' (optionally one), '1..*' (one or many), and '*' (any number, zero or many). For the purpose of hardware modelling, it is useful to allow a more detailed specification of multiplicities, in particular to provide an upper bound 'n' instead of '*'.

User-defined associations have a user-defined interpretation, particularly regarding roles, but no meaning specified in the UML. Hence, they cannot and need not be considered in the meta-model. However, there are two predefined associations, *composition* and *aggregation*, which contribute to the semantics of a UML model. The UML notation of these associations is shown in figure 5.

Composition, also known as *has-parts* relation, is a binary association between exactly one composed object and any multiplicity of sub-objects.

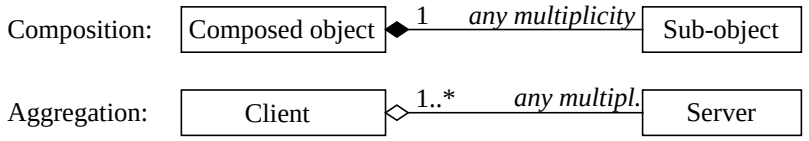


Fig. 5: Associations defined in UML

The sub-objects are exclusively owned and fully encapsulated by the composed object. Any access to the sub-objects must be via the composed object's methods. Hence, the sub-objects can be considered as a part of the composed object's state, i.e., as some of its attributes. In practice, attributes and composition are used interchangeably in OOA/D models. During programming, composed objects are to be implemented as attributes.

For the meta-model, the attribute representation of composition is assumed. A single sub-object becomes an attribute declared with the sub-object's class type. Sub-object multiplicities larger than 1 can be transformed into an attribute which stores an array of objects. Thereby, the sub-objects become an exclusively owned and fully encapsulated part of the state of the composed object. For the purpose of this work, it is not useful to consider them as individual objects. Therefore, sub-objects do not occur in the set of all objects, Ω, while the composed object does.

Aggregation is a relation between aggregating objects (also: *clients*) and their used objects (also: *server*). It is similar to composition with the exception that a used object is not exclusively owned; that is, it can be shared by several clients. Hence, a used object cannot be part of the state of any of its aggregating objects. Instead, it exists on its own, being only referenced by the clients. The possession of such a reference allows the client to communicate with its server. These aspects of aggregation and their integration into our meta-model will be addressed further in section 4.2.

4.1.8 Parameterization and interfaces

Object-oriented implementation languages provide several mechanisms to ease the specification of object models. These are reflected by UML notations for interface inheritance, abstract classes, and class templates with value and type parameters as shown in figure 6.

A class template facilitates the use of parameters in the definition of a class. These parameters can be specified either (a) in a dashed box or (b) by pointed brackets at the place of their use. Variant (a) is preferred in this work

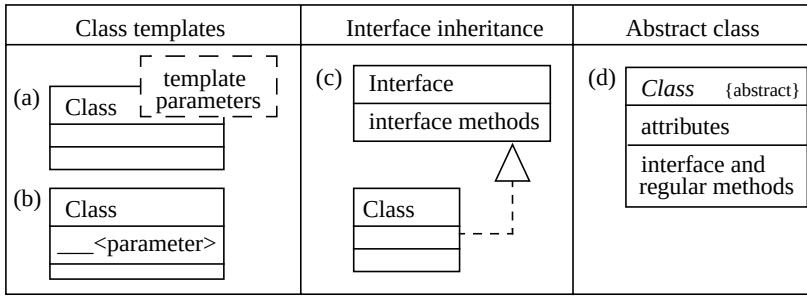


Fig. 6: UML notations for parameterization of classes

because it is more explicit. In the following, template parameters are distinguished into generic values and generic types. Generic values allow to parameterize a value used in the class definition. Generic types provide parameterization of types used in the definition of attributes and method parameters. They can also be used for emulating multiple inheritance by passing additional parent classes as generic types to a derived class. The derived class can instantiate these parents and make their services available to its environment. This modelling style is known as *mix-in inheritance* [30].

An interface (c) allows the specification of interface methods without implementation. Attributes cannot be defined in an interface. A class can inherit not only from a parent class, but also an arbitrary number of interfaces. By inheriting an interface, a class becomes committed to implementing the interface methods. This mechanism can be used as a replacement for multiple inheritance. It resolves the problems related to multiple inheritance of implementations and attributes.

Finally, also an abstract class (d), marked by an italic name and the optional keyword abstract, may include interface methods. However, as opposed to an interface, the abstract class can define regular, implemented methods and attributes, too. Hence, multiple inheritance from several abstract classes can pose problems and is therefore ruled out in this work. A typical application of abstract classes is to factor out common properties of derived classes, while not having to implement all of them. Due to the missing implementations, abstract classes and interfaces cannot be instantiated.

While they may require significant implementation effort, particularly for the parser and semantic analyser, the mentioned parameterization features are no fundamental problem to hardware synthesis. Whenever a class template is instantiated, actual values and types must be provided for the generic values

and types, respectively. If the generics are replaced (at least conceptually) by these actuals, the problem of synthesizing a class template reduces to the problem of synthesizing a class. Interface inheritance commits a class to implementing the interface methods. Semantic analysis has to check this, but a subsequent synthesis step does not need to consider interfaces. Finally, abstract classes have to be dealt with only when building an implementation of a derived class. They do not have to be synthesized themselves since no objects can be created from them.

4.1.9 Example

Having presented a hardware-oriented meta-model of object models and some formal notations to be used in the remainder of this work, it may now be the time for an example to illustrate these concepts. The example will be extended subsequently to cover additional aspects, and it will later serve to demonstrate synthesis.

A central part of the example (see figure 7) is an abstract, non-derived class template, *Buffer*. It is parameterized with the buffer's maximum number of entries, *size*, a generic value which must be a positive integer. Another parameter is the generic type of the items to be stored in the buffer, *Item_t*, which is constrained to be a class type. The buffer has two interface methods, *put* and *get*. The *put* method, when implemented in a non-abstract derived class, accepts an input value of the item type and stores it in a buffer object. The *get* method takes an item out of the buffer and returns it via its output parameter. Since a buffer must be able to store a maximum of *size* entries, it is composed of a respective multiplicity of item sub-objects.

Two classes, *FIFO* and *LIFO*, are derived from *Buffer*. *FIFO* implements the methods so that items are taken out in a first-in first-out order. For this purpose, it has two attributes, *first* and *nxt*, in the integer range from 0 to *size*–1, and a boolean attribute, *empty*. Note that the template parameters are visible in the derived class, too, making it a class template effectively. *LIFO* implements a last-in first-out buffer, i.e., a stack, and has an attribute, *index*, in the range of 0 to *size* as stack pointer. More aspects of the implementation of functionality are addressed later.

Two further classes, *Producer* and *Consumer*, are defined. Each of them aggregates one buffer object. On the other hand, a buffer object may be used by one or more producers and one or more consumers. These multiplicities do not include zero since a buffer with no producer or no consumer would make no sense.

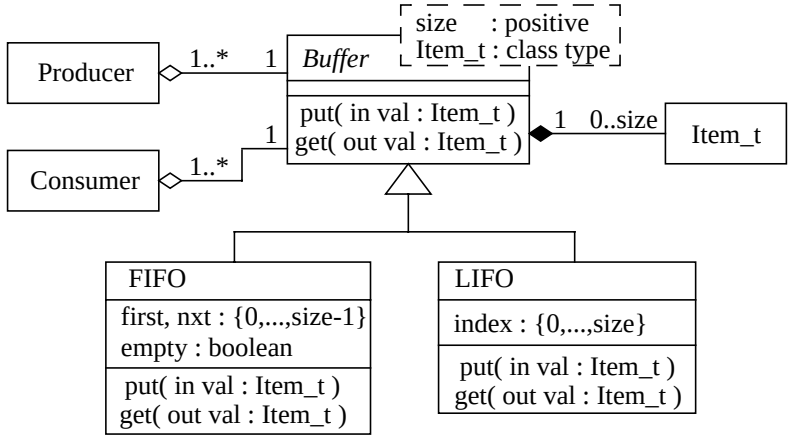


Fig. 7: UML model of parameterized buffers with clients

The following state spaces are defined by the classes:

- $S(Buffer) = S(Item\_t)^{size}$ since *Buffer* has an array attribute with *size* elements of type *Item_t*, representing the composition of 0..*size* item sub-objects (interpreting surplus array elements as unused).
- $S(FIFO) = S(Buffer) \times \{0, \ldots, size-1\}^2 \times \{\text{false, true}\}$ since *FIFO* extends *Buffer* with two attributes with state space $\{0, \ldots, size-1\}$ and one attribute of type boolean whose state space is $\{\text{false, true}\}$.
- $S(LIFO) = S(Buffer) \times \{0, \ldots, size\}$, analogous.

Note that these state spaces are parameterized with the generic value, *size*, and the generic type, *Item_t*.

The methods of *FIFO* have the following input and output spaces:

- *put*: $S(in_{FIFO,\,put}) = S(Item\_t)$, $S(out_{FIFO,\,put}) = \emptyset$ (empty)
- *get*: $S(in_{FIFO,\,get}) = \emptyset$, $S(out_{FIFO,\,put}) = S(Item\_t)$

Input and output spaces are analogous for the methods of *LIFO*. Note that no input and output spaces have been extracted from the interface methods of the abstract class *Buffer* since these are only syntactical artifacts without implementation.

The resulting overall input spaces are the same for *FIFO* and *LIFO*. The same holds for the overall output spaces:

- $I_{Buffer} = I_{FIFO} = I_{LIFO} = S(Item\_t) \cup \emptyset = S(Item\_t)$
- $O_{Buffer} = O_{FIFO} = O_{LIFO} = S(Item\_t) \cup \emptyset = S(Item\_t)$

The method sets are:

- $M_{FIFO} = \{put_{FIFO}, get_{FIFO}\}$ and

- $M_{LIFO} = \{put_{LIFO}, get_{LIFO}\}$.

After splitting these methods into individual next state and output functions, the following overall next state and output functions meeting the condition of Equations 11 and 12 can be defined for *FIFO* (*LIFO* analogous):

$$(15) \quad next_{FIFO}(q, m, inp) = \begin{cases} next_{FIFO,\,put}(q, inp) & if \quad m = put_{FIFO} \\ next_{FIFO,\,get}(q, inp) & if \quad m = get_{FIFO} \end{cases}$$

$$(16) \quad outp_{FIFO}(q, m, inp) = \begin{cases} outp_{FIFO,\,put}(q, inp) & if \quad m = put_{FIFO} \\ outp_{FIFO,\,get}(q, inp) & if \quad m = get_{FIFO} \end{cases}$$

4.2 Communication

Having investigated single objects, we shall now consider the communication between them. We first introduce the object-oriented communication mechanism of message passing and classify different variants thereof. After discussing the interplay of communication with concurrency and synchronisation, communication inside of an object and with an external server object is addressed in detail. Finally, we deal with the situation of an object that can receive messages from multiple concurrent clients.

4.2.1 Message passing

Exchange of messages is the object-oriented way of communication between objects. The term *message passing* has been coined for this mechanism. Although there are some similarities, object-oriented message passing should be told apart from the notion of message passing found in multiprocessor systems [71] and specification languages for concurrent systems such as HardwareC [63] and C* [49]. In the latter contexts, the term denotes the exchange of data via well-defined communication primitives such as send and receive operations, as opposed to data exchange via shared memory. In the object-oriented sense, message passing goes beyond data communication by also transferring control to and by initiating actions of the receiver of a message. The following considerations relate to this object-oriented concept.

By sending a message to a server object, a client object requests the execution of a method from the server. The message includes identification of the requested method and values for its input parameters. When the server object has performed the requested service, the values of the output parame-

ters (if any) have to be transmitted back to the message sender. Using the terminology established in section 4.1.5, message passing to an object of a class C involves the following steps: First, providing a pair consisting of a method identifier and an input value, $(m, inp) \in (M_C, I_{C,m})$, to the target object's implementation. Second, letting the object implementation perform state transition and output computations. Third, providing the computed value, $outp \in O_{C,m}$, to the message sender. The last step can be omitted if $O_{C,m} = \varnothing$, i.e., the method has no output parameters. If it is not omitted, the transmission of an empty reply can be understood as an acknowledgement of the completion of method execution.

The above description corresponds to the understanding of message passing in OOA/D methodologies, which necessarily leave open many details that depend on the implementation language chosen. The variants of message passing are classified according to the following criteria:

- *identification* of the target object,
- *synchronisation*, and
- *concurrency.*

These dimensions are not orthogonal; that is, not all combinations of alternatives make sense. In the following section, concurrency and synchronisation aspects are addressed. Subsequently, several variants of message passing suitable for a hardware implementation will be developed according the possible relations between message sender and message receiver shown in figure 8. The case that an object sends a message to itself or an exclusively owned sub-object (*intra-object communication*) is covered in section 4.2.3. The communication between different objects, *inter-object communication*, is discussed in section 4.2.4. The special and most involved case of objects with several concurrent clients is addressed in section 4.2.5.

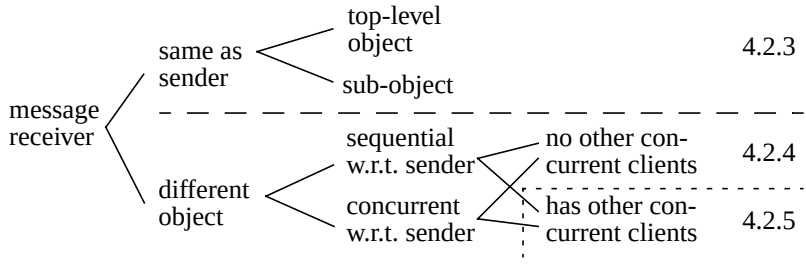


Fig. 8: Variants of object communication

4.2.2 Concurrency and synchronisation

To investigate the potential for concurrency and the need for synchronisation in an object-oriented model, we distinguish concurrency inside a method, concurrency of several method invocations with one object, and concurrency of different objects.

A method has been defined as a subprogram with input and output parameters and access to the object's state. A subprogram, in turn, is a sequential piece of code in most languages. This is necessarily the case in sequential programming languages. The concurrent hardware description language VHDL has strictly sequential subprograms, too. Only few languages, e.g. HardwareC, provide built-in mechanisms to allow sequential code to fork into several concurrent execution threads. In line with the common understanding of object-orientation, a method is therefore considered to be sequential in this work.

Several methods of the same object have all access to the same state, a shared resource. If they were allowed to execute concurrently, the state could become inconsistent. For example, consider a method which modifies an attribute value which has been read before by a second, concurrent method. If the second method performs computations with the meanwhile invalidated value and writes the result back to the attribute, the attribute's state becomes inconsistent. Several mechanisms exist to provide concurrency control. These include Dijkstra's semaphores [43], Hoare's monitors [66], critical regions [31] and path expressions [35]. Most widely, the monitor concept has been adopted. Java's synchronised classes, Ada's protected objects, and the draft standard for controlling access to shared variables in VHDL [165] are all based on a monitor concept. A monitor controls a set of operations so that no more than one can be active at the same time. In other words, *mutual exclusion synchronisation* of the operations is performed. We adopt this concept by defining monitor properties for objects. Hence, one object's methods cannot be invoked concurrently.

While each object itself is internally sequential, different objects can be concurrent with respect to another. Each object has its individual state and encapsulates it completely. Hence, concurrency does not give rise to shared state conflicts. Furthermore, the message communication decouples the execution of different objects' methods from another. A method of one object sends a message to another object to request the execution of a method. Conceptually, both objects can operate concurrently. This must be told apart from the implementation of message passing in sequential OOP languages such as

C++, where only a single thread of computation exists and all object communication is implemented as subprogram calls. Issues related to message exchange between objects are investigated further in the following sections.

4.2.3 Intra-object communication

Communication inside an object occurs when a method of the object requests a service from the object itself or one of its exclusively owned sub-objects. As explained in the previous section, all activity inside an object is sequential. Note that, even though the sub-objects have individual states, their methods cannot be modelled to operate concurrently because only a single thread of computation exists in the complete object. Hence, concurrency does not exist and there is no need for further mutual exclusion synchronisation inside an object. When a message is passed internally within an object, control is transferred to the invoked method. Upon completion of the service, control is returned to the invoking method.

It remains to be answered how the message receiver is identified. A sub-object can be identified by the attribute which implements the sub-object. If this attribute stores an array of objects, an indexing operator of the implementation language can be used to identify a particular object.

Two mechanisms are defined for an object to address itself. First, the object itself is assumed by default if no receiver is specified. Second, a self-reference can be used if provided by the implementation language. For example, self-referentiation is facilitated by the keyword **this** in C++ and Java. The two alternatives are differentiated later when dynamic binding is discussed (see section 4.3.4).

Be $t \in Types$ the class type of the object that receives the message, $m \in M_t$ a method of this class, $inp \in S(in_m)$ a value from the method's input space, and $ret \in S(out_m)$ a value from its output space. The following notations are defined to capture intra-object communication:

- $(attribute, (m, inp), ret)$ indicates that method *m* of the sub-object represented by $attribute$ is invoked with the input parameters *inp* and returns the output parameters *ret*.
- $(\varepsilon, (m, inp), ret)$ denotes the invocation of method *m* of the object itself.
- $(self, (m, inp), ret)$ means using the self-reference for invoking the method *m*.

Figure 9 illustrates the transfer of control (in the sense of possession of a control thread) and data involved by the method invocation described above.

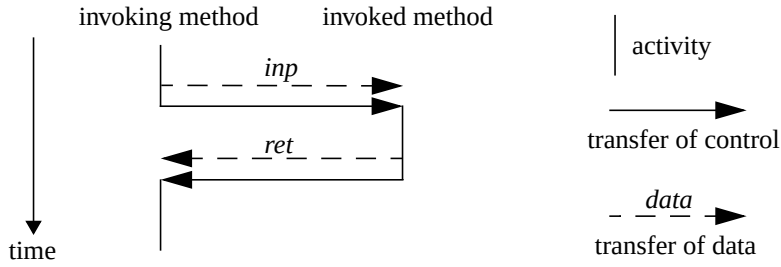


Fig. 9: Control and data transfer during method invocation

4.2.4 Inter-object communication

In inter-object communication, the receiver of a message is no longer a part of or identical to the sender, as it is the case with sub-objects or an object itself. Hence, when considering communication between different individual objects, we must clarify how the sender gets to know the receiver. The answer lies in the aggregation relationship, which makes a server object known to a client. Note that the server is not exclusively owned by the client; it may have other clients as well (cf. section 4.1.7).

Several similar mechanisms for expressing aggregation exist in OOP languages. An aggregated object can be identified by an address variable, reference, or handle. An *address variable* (pointer) denotes a memory location where an object can be found. It can be manipulated using arithmetic operators. A *reference* is an address variable without address arithmetics. Finally, *handles* abstract from memory locations. Thereby, they enable object communication in distributed systems.

Pointer, reference, and handle values can be passed at run-time from object to object, e.g., as method parameters, and they can be created dynamically. This is necessary since their designated objects are subject to dynamic allocation. In this work, however, a static set of objects is assumed. This allows to express aggregation statically, too. Thereby, the problems related to the synthesis of pointers (cf. [147]) are avoided.

The static representation of the aggregation of a client and a server is called a *channel* in this work. The static set of all channels, ζ, is:

(17) $\zeta \subseteq \Omega \times \Omega$.

A channel is a pair of objects, the client and the server:

(18) $ch = (client, server) \in \zeta$.

Be $t \in Types$ the class of the *server* object, $m \in M_t$ a method of this class, $inp \in S(in_m)$ a value from the method's input space, and $ret \in S(out_m)$ a value from its output space. The notation

(19) $(ch, (m, inp), ret)$

denotes message passing via channel *ch* to request the execution of method *m* with input parameters *inp* from the server, and the transmission of the resulting output parameters, *ret*, to the client.

For discussing concurrency and synchronisation aspects of this message exchange, it must be distinguished whether the client and the server are concurrent or sequential with respect to another. In the latter case, their methods are executed by a single thread of computation. Hence, invocation of the server's method from a method of the client includes the transfer of control as in the previous section (see figure 9). Yet, mutual exclusion synchronisation can be required. This is the case if the server object has other, concurrent clients. This issue is addressed in the following section.

If client and server are concurrent, their respective methods are executed by different threads of computation. Hence, message passing is reduced to the transfer of data, but not control. This gives rise to the question whether a client object can continue its operation while the server executes the requested service. However, this is not possible in general because message passing includes the sending of a request and the reception of the results in a single action. Hence, as illustrated in figure 10, the client object must wait for the results from the server before it can resume.

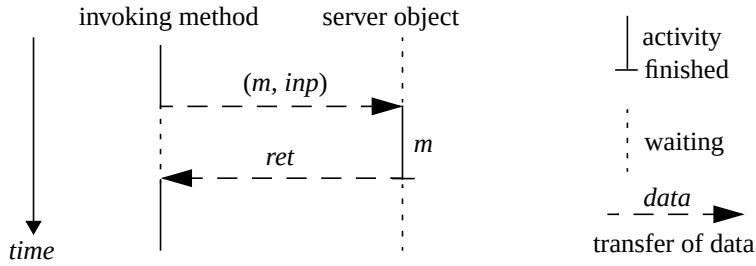
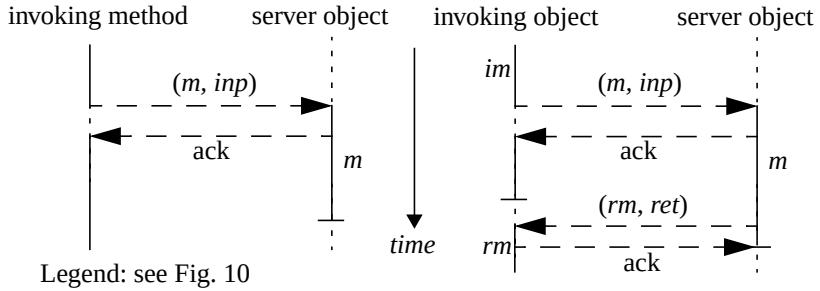


Fig. 10: Message passing between concurrent objects

An exception to this blocking behaviour of message passing can be made if the requested service does not have output parameters. In this case, after sending the method identifier and the input parameters, the invoking method could resume. In this work it is assumed that it waits for acknowledgement of

message receipt before resuming (see left hand of figure 11). Otherwise, the client would be able to issue another message before the first one has been accepted. This could cause inconsistencies or loss of the first message unless a queuing mechanism exists.

Note that the user can apply a modelling style for asynchronous message passing using only messages without output parameters. To transmit results, a server would have to send a return message, *rm*, to its client, as shown on the right hand of figure 11.



Legend: see Fig. 10

Fig. 11: Message passing without output parameters

As we have seen, the potential for concurrency between *two* communicating objects is limited. However, a server can have multiple clients. In this case, which is considered in the following section, concurrent activities may exist in the objects even if message passing includes transmission of output parameters.

4.2.5 Concurrent service requests

While the previous discussions have been focused on the client of a single client-server relationship (aggregation), we now take the view of a server which has multiple clients. Figure 12 shows two situations in which a server, *S*, receives requests from two clients, *C*1 and *C*2. On the left hand, the second service request arrives after the first one has been completed. On the right hand, a second request is issued while the server still is still busy serving another client. As previously defined, mutual exclusion ensures that only a single method is under execution at a time. Hence, the second service request is put on hold until the first one has been completed. Note that concurrency exists at a given time between two objects if they are not in the course of mutual communication.

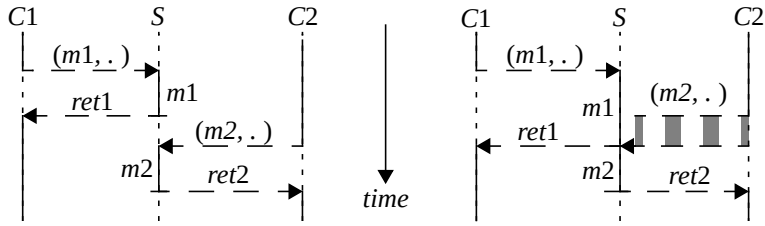


Fig. 12: Server receiving messages from concurrent clients

However, mutual exclusion may not be sufficient. Even if a server is not busy, it may be unable to service a request because it lacks other resources. For example, a buffer which is full cannot accept a *put* request. Such conditional acceptance of requests is referred to under the term *condition synchronisation*. Several techniques for modelling condition synchronisation have been developed in COOP research (cf. section 2.4.3). These include:

- Object bodies which allow to model explicitly the acceptance of messages and the invocation of the corresponding methods [112].
- Guarded methods which specify the conditions under which they can be invoked as a guard expression [53][145]. Message acceptance and method invocation are performed by the run-time system.

In this work, we abstract from such particular modelling mechanism. The intention is to capture synchronisation conditions in an abstract way, regardless of the modelling philosophy.

Whether or not a message $(ch, (m, inp), ret)$ can be accepted by the server object of some class type, $t \in Types$, can depend on the following information:

- The state of the server object, $q \in S(t)$.
- The requested service, $m \in M_t$.
- The input parameter values of the service, $inp \in I_t$.
- The history of previous service requests, which can be encoded as part of the object's state by the user following Schumacher's modelling style [144][145].

Hence, method acceptance is described by the synchronisation condition (guard) function,

(20) $\quad g_t : S(t) \times M_t \times I_t \rightarrow \{\text{false}, \text{true}\}$,

where $g_t(q, m, inp)$ = true if method m can be invoked with the input parameters *inp* when the object is in state q, and $g_t(q, m, inp)$ = false otherwise.

Service acceptance may be independent of the input parameter values. If this is the case for all services of the class *t*, the synchronisation condition function can be simplified as follows:

(21) $g_t : S(t) \times M_t \rightarrow \{\text{false}, \text{true}\}$.

If a service cannot be accepted at some time, the client must be put on hold until the object's state has changed so that the service is acceptable. A state change can only be caused by the execution of a service requested by another, concurrent client[1]. This situation is illustrated in figure 13. Client *C*1 sends a message *m*1 to the server, *S*. The message is not accepted in the server's current state. Suppose that another message *m*2 is issued by client *C*2. This may cause a state transition of the server after which the first message, still with the same input parameters, becomes acceptable. Hence, after the execution of the service *m*2, the first message can be accepted. Note that condition synchronisation has caused a re-ordering of services. The service requested first is executed after the other service which has been requested later.

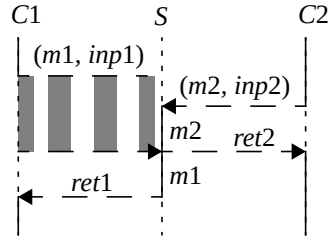


Fig. 13: Condition synchronisation of service requests

4.2.6 Example

Concurrency aspects shall now be investigated for the example of section 4.1.9. In the example (cf. figure 7), a *Buffer* is aggregated by a *Producer* and a *Consumer*. It is assumed that these two clients and their common server, *Buffer*, are concurrent. Note that the actual server must be either a *FIFO* or a *LIFO* object since no objects of the abstract class *Buffer* can be instantiated[2]. The following considerations are for a *FIFO*; *LIFO* is analo-

1. This is why condition synchronisation is relevant only in a concurrent context.

2. The capacity of a derived object to take on tasks of a parent class is related to the concept of polymorphism presented in section 4.3.

gous. Following section 4.2.2, mutual exclusion is defined for the services of *FIFO*. Condition synchronisation is required, too. A *FIFO* whose places for storing item objects are all occupied cannot accept a *put* request. An empty *FIFO*, in turn, cannot accept a *get* request. These synchronisation conditions are independent of the input parameters of these services. Hence, a guard function can be defined as:

$$(22) \quad g_{FIFO}(q, m) = \begin{cases} \neg is\_full(q) & if \;\; m = put \\ \neg is\_empty(q) & if \;\; m = get \end{cases}$$

A *FIFO* with *put* and *get* services for storing and retrieving several items at a time can serve as an example for input-dependent synchronisation conditions. It is assumed that the number of these items is part of the input space of the object. In this case, a *put* service can be performed only if sufficient free places exist for the number of items to be stored. A *get* service is accepted only if the *FIFO* contains at least the number of items to be retrieved.

If a *get* request from a *Consumer* is not acceptable, this request is blocked until one or several *put* requests from a *Producer* have sufficiently filled the *FIFO*. Vice versa, acceptance of a blocked *put* request is enabled after *get* requests have taken enough entries out of the buffer.

Finally, a sequential aspect of the object model shall be mentioned. The exclusively owned item sub-objects of a *Buffer* are sequential with respect to each other and to the *Buffer* according to the considerations of section 4.2.3 on intra-object communication. No mutual exclusion or condition synchronisation is required for the sub-objects.

4.3 Polymorphism

Having explained the communication between objects, we can now deal with polymorphism of objects and dynamic binding of service requests. We present these concepts as available in programming languages first. For a hardware implementation, a different interpretation and implementation is required. This leads to the concepts of value-based polymorphism and channel switching. We detail dynamic binding in the framework of our metamodel and finally present a refined finite state machine model that covers the new concepts.

4.3.1 Polymorphism mechanisms

Polymorphism allows to send messages to an object whose class membership is not or not exactly known. Moreover, it facilitates the run-time change of the object addressed by a message passing operation. This object change can occur between successive invocations of the operation. In consequence, the class membership of the object addressed may change, too.

As an extreme case, a language may allow to send any message to any object, regardless of whether or not the object implements a corresponding service. If it does not, a run-time error occurs. Such un-typed polymorphism can be found in Smalltalk [98] and in the Vista OO-VHDL dialect [37].

In this work, a variant of polymorphism called *inclusion polymorphism* is assumed. Inclusion polymorphism uses type information about the server object, exploits properties of method inheritance, and imposes restrictions on the messages that can be sent. By this combination, safe message passing can be ensured; that is, message passing which by construction cannot lead to run-time errors. Similar mechanisms are the basis for polymorphism in Ada 95 and Java.

While the exact class membership of a server object may not be known statically and may even change at run-time, we demand that the object is of a given *root class*, *R*, or of any class derived from *R*. Note that *R* can be a non-derived class or a derived class; the term *root* refers to *R*'s role as root of the inheritance subtree of classes involved in polymorphism (see figure 14). In the UML, the use of a class in an association implicitly includes all its subclasses. For instance, while the *Producer* and *Consumer* in figure 7 aggregate a *Buffer*, the actual object aggregated can be a *FIFO* or a *LIFO* derived from *Buffer*.

The methods, $m \in M_R$, of the root class include the methods defined in the class itself and all inherited methods (if *R* is a derived class). All these methods are, in turn, inherited by the classes derived from *R*. It is therefore safe to request these services from an object of class *R* or derived. However, all other services, particularly those which are added in *R*'s subclasses, are unavailable at least from *R*. Hence, in the inclusion polymorphism framework, it is permitted to request the methods, $m \in M_R$, and only these, from an object of root class *R*. Figure 14 illustrates the locations where these methods are defined.

A method can be redefined in a derived class. Thereby, the functionality of a service can be adapted to the purpose of the derived class. When such a service is requested from a polymorphic object, its execution must be dis-

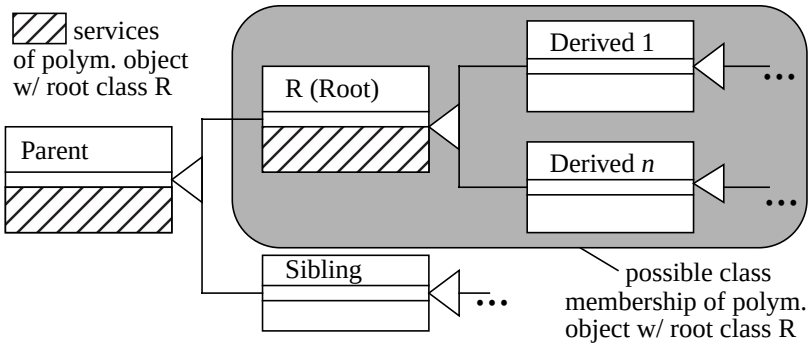


Fig. 14: Structural aspects of polymorphism

patched to the version of the redefined method that is valid for the object's current class. This has to be done at run-time by a mechanism called *dynamic binding* since the object's class membership is not known statically.

The root class of a polymorphic object can be abstract. In this case, some of the methods may be abstract, i.e., not have an implementation. However, these methods must be implemented in all non-abstract derived classes. Furthermore, no object of an abstract class can be created. Hence, we can be sure that dynamic binding always finds a suitable method implementation.

4.3.2 Polymorphic objects

In OOP languages, the use of pointers, reference variables, or handle variables allows to designate objects whose class membership is statically unknown. The assignment of a new value to one of these causes a change of the designated object. As a result of that, also the class of the designated object can change.

In the meta-model, static channels are defined (cf. section 4.2.4). A channel designates a server object that can be accessed by a client object. The server object to which a channel connects has been defined to be invariable. This implies that, in order to obtain a variable class membership of a server object, the object itself must be able to change its class membership. We call such an object *polymorphic* in the following.

The state space of a polymorphic object with root class *R*, $S(R_{poly})$, comprises the state space of *R*, as far as *R* is non-abstract, and all the states of the non-abstract classes derived from *R*. In addition to representing these states, it must be possible to determine the class membership of the polymor-

phic object for dynamic binding. However, two states from different classes are not always distinguishable. For instance, the states of a derived class which adds no attribute are absolutely the same as the states of the class' direct parent. Hence, additional class information is provided by extending the state space with a class identifier:

$$(23) \quad S(R_{poly}) = \dot{\bigcup_{c\ is\text{-}a^*\ R \wedge \neg abstract(c)}} \{c\} \times S(c) \subseteq Types \times Values.$$

Assume that two different classes, t_1 and t_2 where $t_1 \neq t_2$, share a state, $q \in S(t_1)$ and $q \in S(t_2)$. In the polymorphic state space, the state is distinguished by the class identifier: $(t_1, q) \neq (t_2, q)$. Hence, the union in equation 23 is disjoint.

A polymorphic object, $obj \in R_{poly}$, can have a state $q_{R,\,poly} = (id, q)$ of any non-abstract class derived from *R*, including *R*. Over time, the object can switch class membership. The following properties always hold:

- A class c derived from *R*, $c \in Types$ with $c\ is\text{-}a^*R$, exists so that $obj \in c$, i.e., class membership is constrained to the classes allowed by inclusion polymorphism.
- $obj \in c \Leftrightarrow id = c$, i.e., the class identifier always indicates the correct class membership.
- $id = c \Rightarrow q \in S(c)$, i.e., the polymorphic object always has a state of the class indicated by its class identifier.

Note that, while the set of all objects, Ω, remains static, individual objects can switch between the classes whose union is Ω (cf. equation 14).

A polymorphic object can be an exclusive sub-object of another object. In this case, it is designated by an attribute name instead of a channel. It does not appear as an individual object in Ω. Still, the extension for a class identifier applies to the sub-object's state space.

A change of a polymorphic object's class membership is caused by an assignment. This requires the implicit definition or explicitly modelling of an assignment operation (method). Restrictions may apply on the classes between which an assignment is permitted. This depends on the target language and will be addressed for Objective VHDL in the next chapter. Moreover, chapter 6 will address optimizations related to polymorphic objects.

The concepts presented are similar to variant records in Pascal [107] and unions in C/C++ [151]. Differences are in the automatic management of a discriminant (in the form of the class identifier) and the integration with a dynamic binding mechanism provided for polymorphic objects.

4.3.3 Channel switching

Another approach to polymorphism which is feasible for a hardware implementation is *channel switching*. Imagine that a client sends over a channel, *ch*, a message to a polymorphic intermediate object (pseudo object) with root class *R*. This object be connected via individual channels, ch_1, ..., ch_n, to a set of server objects (including, but not necessarily, polymorphic objects) each of which is of class *R* or derived. If the pseudo object forwards an incoming message to one of these channels, the message arrives at a server object whose class membership is not known to the client. From the view of the client, this is polymorphism. Figure 15 illustrates this scenario.

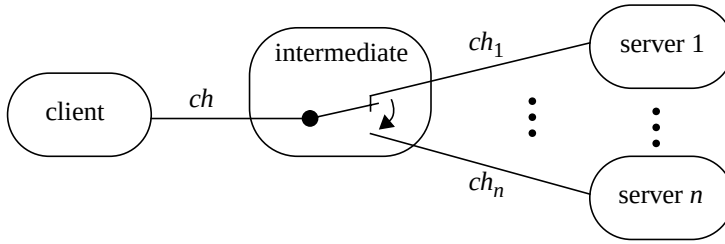


Fig. 15: Channel switching

Channel switching becomes relevant when an object is passed as a parameter of a method. The formal parameter can be understood as the intermediate object of figure 15. At different invocations of the method, different server objects are passed as actual parameters. Hence, messages sent via the intermediate object arrive at different server objects. Note that, although parameter passing is a run-time activity, objects and channels are still static. Only the association of formal and actual channel, which is represented as a channel switch, is dynamic.

4.3.4 Dynamic binding

Following the inclusion polymorphism concept, the services specified in class *R*, and only these, can be requested from a polymorphic object with root class *R*. Hence, we define the set of methods that can be invoked with such an object as:

(24) $M_{R,\,poly} = M_R$.

Assume that a method, $m \in M_{R,\,poly}$, is not redefined in classes derived from *R*. In this case, a corresponding service request is executed by the same method, *m*, regardless of the current class membership of the object. Method invocation is said to be *statically bound*.

The opposite, *dynamic binding* (cf. section 4.3.1), can be defined as follows. We assume that a message, *m*, is sent to a polymorphic object, $obj \in R_{poly}$, which may be a server object in an aggregation or an exclusive sub-object. Be $c \in Types$ the class to which the object belongs at the time of service invocation. The service is executed by the method's latest redefinition with respect to the class *c*. This method version, m_{latest}, is defined as follows:

- m_{latest} is a redefinition of *m*, defined in a class *d* which is
  – derived from the root class *R* of polymorphism and
  – direct or indirect parent of the polymorphic object's current class, *c*.
- There exists no other redefinition of *m* in any class *t* which is between *d* and *c* in the inheritance relationship (including *c*, but excluding *d*).

This is reflected in the formal definition: The latest redefinition of a method *m* with respect to class *c* is

(25) $m_{latest} \in M_d$ where $d\ is\text{-}a^*R$ and m_{latest} *redefines m* and
$\neg\exists t \in Types : t\ is\text{-}a^*d \wedge c\ is\text{-}a^*t \wedge t \neq d \wedge \neg\exists m_{redef} \in M_t :$
m_{redef} *redefines m.*

Note that this definition includes the following corner cases:

- $R = d$ (and $m = m_{latest}$), i.e., there exists no redefinition of *m* in the branch of inheritance from *R* to *c*. However, redefined methods requiring dynamic binding may exist in other classes derived from *R*.
- $d = c$, i.e., the latest redefinition of *m* is in the current class itself.

Figure 16 gives an overview of the classes and methods involved in the above definition, and of their relationships.

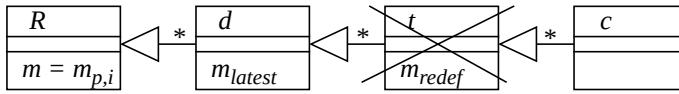


Fig. 16: Class relationships and method redefinitions in equation 25

Up to now, requests to a polymorphic object have been considered. A sort of dynamic binding can occur when an object sends a message to itself, too, even if it is not polymorphic. Assuming that the message is sent from an inherited method and that the service requested has been redefined in the course of inheritance, two cases are differentiated (cf. section 4.2.3):

- The message is sent to the ε reference. In this case, the original method of the parent class is invoked also by the derived object (static binding).
- The message is sent to the *self* reference. In this case, the latest redefinition of the method is invoked to execute the service (dynamic binding).

4.3.5 FSM representation of polymorphic objects

The behaviour of a polymorphic object can be captured by defining an equivalent finite state machine in a way similar to section 4.1.5 from which we use the definition of *next* and *outp* functions for each method (cf. equation 10). These functions be adapted to the input and output spaces of the root class of polymorphism, *R*; that is, *R* corresponds to *C* in section 4.1.5. The functions still have to be adjusted to the state space of the polymorphic object and to be combined into a single next state function and output function, respectively. These global functions shall take dynamic binding into account.

This is done by defining $next_{poly} : S(R_{poly}) \times (M_R \times I_R) \rightarrow S(R_{poly})$ and $outp_{poly} : S(R_{poly}) \times (M_R \times I_R) \rightarrow O_R$ so that for all methods, $method_{p,i} \in M_R$, for all inputs of the method, $r \in S(in_{p,i})$ and for all states, $q_{R,poly} \in S(R_{poly})$, the following holds:

Be $q_{R,poly} = (c, q_c)$, be $method_{d,j} \in M_d$ the latest redefinition of $method_{p,i}$ with respect to the object's current class, *c*, and be $q_c = (q_d, q_{ext})$. The global functions fulfil:

(26) $next_{poly}(q_{R,poly}, (method_{p,i}, r)) = (next_{d,j}(q_d, r), q_{ext})$ and

(27) $outp_{poly}(q_{R,poly}, (method_{p,i}, r)) = outp_{d,j}(q_d, r)$.

The relationship between *R*, *c*, and *d* is as in figure 16. The polymorphic state is first split into the class identifier, *c*, and a state of this class, q_c. Then, the latest redefinition of the requested method is determined with respect to *c*. The state is further split into the portion, q_d, which belongs to the class, *d*, in which the latest redefinition has been found, and the state extensions, q_{ext}, added in the inheritance steps towards *c*. Only q_d is taken into account by the functions while q_{ext} is ignored and remains unmodified. Note that the latest redefinition of each method can be pre-computed statically for all possible classes, *c*, in order to avoid performing this selection process in hardware.

With the above definitions, a polymorphic object with root class *R* can be implemented by the finite state machine

(28) $FSM_{R,poly} = (S(R_{poly}), (M_R \times I_R), O_R, next_{poly}, outp_{poly}, init)$.

4.3.6 Example

To illustrate polymorphism and dynamic binding, the buffer example is considered again. We explain the properties of a polymorphic object with root type *Buffer*, $obj \in Buffer_{poly}$. Note that a polymorphic object can be created even if its root class is abstract. It cannot take on the state and class membership of the abstract class, *Buffer*. However, it can take on the state and class membership of any derived class, i.e. *FIFO* or *LIFO*. The specific class membership may change over time.

The state space of the polymorphic buffer object, *obj*, is the union of the state spaces defined by the non-abstract classes *FIFO* and *LIFO*, each augmented with a class identifier:

(29) $S(Buffer_{poly}) = \{FIFO\} \times S(FIFO) \cup \{LIFO\} \times S(LIFO)$.

The services available from *obj* are *put* and *get*, which are defined for *Buffer*. Should the derived classes define additional services, these could not be invoked with *obj*.

(30) $M_{Buffer,\, poly} = M_{Buffer} = \{put_{Buffer}, get_{Buffer}\}$.

As it has already been mentioned, *obj* can change between *FIFO* and *LIFO* class membership. The methods, *put* and *get*, are implemented differently for these classes. Hence, dynamic binding is applied. A *get* request is dispatched to the implementation of *get* for *FIFO* if $obj \in FIFO$, and to *get* for *LIFO* if $obj \in LIFO$. The analogue holds for *put*. A state transition function of the polymorphic object FSM can be defined as follows; the output function is analogous (replace *next* by *outp*):

$$next((c, q_c), (m, inp)) =$$

$$(31) \quad \begin{cases} next_{FIFO,\, put}(q_c, inp) & if \quad c = FIFO \wedge m = put_{Buffer} \\ next_{LIFO,\, put}(q_c, inp) & if \quad c = LIFO \wedge m = put_{Buffer} \\ next_{FIFO,\, get}(q_c, inp) & if \quad c = FIFO \wedge m = get_{Buffer} \\ next_{LIFO,\, get}(q_c, inp) & if \quad c = LIFO \wedge m = get_{Buffer} \end{cases}$$

4.4 Hardware implementation of objects

In order to implement an object as a digital circuit, we must define a binary encoding of its state space. This is done first for objects of a non-derived or derived class, and subsequently for polymorphic objects. Likewise, the mes-

sages an object can receive must be encoded, for which purpose the polymorphic techniques can be adapted.

After defining these encodings, we devise an implementation of non-derived objects first. Two alternatives, delegation and expansion, are presented for the implementation of derived objects. While the methods are treated as black boxes in our approach, to be synthesized by a HLS back-end, we take a look into potential optimizations that could be applied. A polymorphic object can be synthesized in a way similar to a non-polymorphic one under consideration of its different state space and encoding and with special structures for dynamic binding.

4.4.1 Encoding of state spaces

For implementing an object as a digital circuit, a binary encoding of the object's state space must be chosen. This encoding is an injective mapping from the state space into the set of bit-vectors of length (number of bits) $b(C) \in \mathbb{N}$:

$$(32) \quad enc_C : S(C) \rightarrow \{0, 1\}^{b(C)}.$$

The length must be chosen sufficiently large to allow enc_C to be injective; that is, to map no two different states to the same encoded value. Since there are $2^{b(C)}$ different bit-vector values of length $b(C)$, we must demand that the number of states is smaller or equal:

$$(33) \quad |S(C)| \leq 2^{b(C)}, \text{ which yields } b(C) \geq \lceil \log_2(|S(C)|) \rceil.$$

Note that the use of the shortest possible encoding length would imply that attributes of the object-oriented description might have to share state bits. For instance, a state space defined by an attribute with three values and a second attribute with five values comprises a total of $3 \cdot 5 = 15$ states, which can be encoded with four bits. It is, however, desirable to encode each individual attribute with its own set of bits, so as to ease the extraction of individual attribute values. This means to encode the first attribute with two bits and the second one with three bits, yielding a total of five bits.

Furthermore, it may not be desirable to encode an attribute with a minimum of bits. For example, in order to allow an efficient implementation of arithmetic operators, a four-bit two's complement must be used to encode integer values from –1 to 6, even if a minimal encoding of these eight values would only require three bits.

The attribute-wise encoding is achieved by defining the encoding of a state, q_C, of a non-derived class, *C*, as the concatenation of the encodings of the attribute values, $q_{t_{C,i}}$:

(34) $enc_C(q_C) = (enc_{t_{C,1}}(q_{t_{C,1}}), \ldots, enc_{t_{C,n(C)}}(q_{t_{C,n(C)}}))$.

The encoding length follows as the sum of the attributes' encoding lengths:

(35) $b(C) = \sum_{i=1}^{n(C)} b(t_{C,i})$.

For a derived class, *D*, the state encoding is again attribute-wise. In addition, the part of the state space inherited from the direct parent class, *P*, is encoded with the same encoding as in *P*:

(36) $enc_D(q_D) = (enc_P(q_P), enc_{t_{D,1}}(q_{t_{D,1}}), \ldots, enc(q_{t_{D,n(D)}}))$

The resulting encoding length is the sum of the parent's encoding length and the encoding lengths of *D*'s attributes:

(37) $b(D) = b(P) + \sum_{i=1}^{n(D)} b(t_{D,i})$.

Note that the above definitions are recursive in two ways:

- The encoding of a derived class is defined with the help of its direct parent's encoding. This parent may again be a derived class. Recursion ends if a non-derived class is reached, which is always the case thanks to the acyclic tree structure of the inheritance relationship (cf. section 4.1.4).
- Class encodings use the encoding of attributes. There is no recursion if an attribute is of a primitive type[3]. However, an attribute may also be declared with a class type so as to represent an exclusive sub-object. It must be demanded that the sub-object does not include, directly or indirectly, the state of the composed object. This implies, in particular, that the sub-object must not be derived from or of the same class as the composed object, and that the sub-object must not have another sub-object for which this is the case.

The limitations described ensure that recursion always advances upwards the inheritance tree towards a non-derived class. Under this condition, in the absence of cycles, the recursive encoding is well-defined.

3. Implementation languages provide different choices of primitive types. In this work, those of VHDL and their encoding are discussed in chapter 7.

4.4.2 Polymorphic objects

When encoding the state space of a polymorphic object, the class identifier must be encoded in addition to the attributes. As in the case of the attributes, it is desirable to encode the class identifier with an individual set of bits. This allows to extract the class membership information of a polymorphic object more easily compared to minimal encoding. Two main concepts exist for this purpose: virtual tables and tagged types.

Virtual tables are used by C++ compilers to implement dynamic binding of virtual functions (dynamically bound methods) [72]. Each object of a class has a pointer to the virtual table of its class. This pointer can be understood as a class identifier. Its dereferentiation yields the virtual table which contains pointers to the latest redefinition of each virtual method of the class. After looking up the address of the latest redefinition of a method, this address is used for calling the method.

The virtual table approach has two important drawbacks. First, it implies that virtual methods are always dynamically bound, even if the class membership of an object is statically known so that static binding could be applied. Many performance concerns against the use of OOP languages stem from this inherent overhead. Secondly, it is based not only on pointer dereferentiation but also on calling of pointer-addressed subprograms. Hence, the concept is hard to implement with languages, such as VHDL, that do not support these features. Likewise, implementation is problematic for target systems which do not support pointers well, such as distributed systems without common address space and, of course, digital circuits.

A different concept, *tagged types*, is employed in Ada 95. A tag is an identifier of the class membership of an object, added by the compiler to the object's state space. It facilitates the implementation of dynamic binding without pointers which is the reason for applying tags in this work. Moreover, it allows to apply dynamic binding only when the class membership of an object is not statically known. Otherwise, the method to be invoked can be determined statically, even for redefined (virtual) methods. If dynamic binding is not used at all with an object, even its tag can be omitted.

Since the tag, by indicating class membership, allows to tell apart states of different classes, these can be mapped to the same encoded representation. Only the states of each individual class need to be distinguished. This can be achieved by choosing the same state encoding as for a non-polymorphic object. Hence, we define the binary encoding of a polymorphic object with root class R as the injective function

(38) $enc_{R,\,poly} : S(R_{poly}) \rightarrow \{0,1\}^{b(R_{poly})}$,

which encodes each state $q_{R,\,poly} = (c, q_c)$ by the concatenation of the class identifier's encoding and the state encoding for class *c*:

(39) $enc_{R,\,poly}(c, q_c) = (enc_{Tag}(c), enc_c(q_c))$.

Note that an object with a short state vector may leave some bits unused. The encoding length follows as the maximum of the sum of the tag encoding length and the state encoding length of class *c*. The maximum is computed over all classes *c* whose membership can be taken on by the polymorphic object:

(40) $b(R_{poly}) = max\{b_{Tag}(c) + b(c) \mid c\ is\text{-}a^*R \wedge \neg abstract(c)\}$.

This definition would allow to use a short tag encoding for a class with large state space, and a longer tag for a class which needs less state bits. This might help to reduce the overall encoding length of the polymorphic object's state by a few bits. However, non-uniform tag length complicates tag and attribute extraction. In the following, we assume that all tags of a polymorphic object are encoded with the same number of bits. This allows to draw tag length out of maximum computation:

(41) $b(R_{poly}) = b_{Tag}(R_{poly}) + max\{b(c) \mid c\ is\text{-}a^*R \wedge \neg abstract(c)\}$

Since the state space cannot diminish in the course of inheritance, it is sufficient to consider leaf nodes (classes) of the inheritance tree during practical computation of encoding length.

Again, the above definition is recursive. For computing $b(R_{poly})$, the encoding length of class *R* and the classes derived from *R* have to be known. To avoid a cyclic, ill-defined encoding, these classes must have neither directly nor indirectly a polymorphic sub-object with root class *R*.

4.4.3 I/O encoding

The encoding of the input and output spaces of an object's methods is similar to the encoding of state spaces. While a state is comprised of attribute values, the input and output of a method is comprised of the input and output parameter values, respectively. As in the case of attributes, easy access to the individual parameters is desirable. Hence, each input and output parameter is encoded individually. Likewise, a non-minimal encoding may be applied to the individual parameters.

This leads to the following definition of a binary encoding of the input space of a method: Be $method_{c,\,i} \in M_C$ a method of class *C* and

$k = k(method_{c,i})$ the number of the method's input parameters. The input encoding is the mapping

(42) $enc_{in_{c,i}} : S(in_{c,i}) \rightarrow \{0,1\}^{b(in_{c,i})}$, where

(43) $enc_{in_{c,i}}(inp) = \left(enc_{t^{in}_{c,i,1}}(p^{in}_{c,i,1}), \ldots, enc_{t^{in}_{c,i,k}}(p^{in}_{c,i,k})\right)$ and

(44) $b(in_{c,i}) = \sum_{j=1}^{k} b(t^{in}_{c,i,j})$.

A binary output encoding is defined in analogy: Be $method_{c,i} \in M_C$ a method of class C and $l = l(method_{c,i})$ the number of the method's output parameters. The output encoding is the mapping

(45) $enc_{out_{c,i}} : S(out_{c,i}) \rightarrow \{0,1\}^{b(out_{c,i})}$, where

(46) $enc_{out_{c,i}}(outp) = \left(enc_{t^{out}_{c,i,1}}(p^{out}_{c,i,1}), \ldots, enc_{t^{out}_{c,i,l}}(p^{out}_{c,i,l})\right)$ and

(47) $b(out_{c,i}) = \sum_{j=1}^{l} b(t^{out}_{c,i,j})$.

From here on, we identify each method by a unique number, $method_{c,i} \in \{1, \ldots, |M_C|\}$. In addition, the value 0 (zero) is used to identify the absence of a service request on a channel. An encoding of method identifiers maps these values into the set of bit-vectors of the encoding length $b(M_C)$:

(48) $enc_{M_C} : \{0, 1, \ldots, |M_C|\} \rightarrow \{0,1\}^{b(M_C)}$

This mapping can be chosen arbitrarily as long as it is injective. In this work, the method identifier's representation in the dual system is used. Hence,

(49) $enc_{M_C}(m) = m_{(2)}$ and $b(M_C) = \lceil \log_2(1 + |M_C|) \rceil$.

Having defined the I/O encoding of each individual method, we must still consider that any service of an object can be requested via a channel, one at a time. Hence, the input space, I_C, and output space, O_C, of the object are to be represented by a number of bits, $b(I_C)$ and $b(O_C)$, that allow to encode the largest input and output of any method:

(50) $b(I_C) = max\{b(in_{c,i}) \mid m_{c,i} \in M_C\}$,

(51) $b(O_C) = max\{b(out_{c,i}) \mid m_{c,i} \in M_C\}$.

Note that this does not allow to speak of an encoding of the object's input or output space since inputs or outputs of different methods may be mapped to the same bit representation. However, the method identifier enables the correct interpretation of these inputs or outputs. Therefore, the input encoding of an object of class *C* is defined as follows:

(52) $enc_{I_C} : M_C \times I_C \rightarrow \{0,1\}^{b(M_C)+b(I_C)}$ where

(53) $enc_{I_C}(m_{c,i}, inp) = (enc_{M_C}(m_{c,i}), enc_{in_{c,i}}(inp))$.

An analogous encoding is defined for the output of an object of class *C*:

(54) $enc_{O_C} : M_C \times O_C \rightarrow \{0,1\}^{b(M_C)+b(O_C)}$ where

(55) $enc_{O_C}(m_{c,i}, outp) = (enc_{M_C}(m_{c,i}), enc_{out_{c,i}}(outp))$.

The method identifier in the above encodings can be understood as a kind of tag that distinguishes the inputs and outputs of different methods. In this sense, I/O encoding shows a duality with polymorphic encoding. This duality has been exploited in [119][120] by implementing message passing between Objective VHDL entities with the help of polymorphic message objects. In the following, it will allow us to apply similar mechanisms for implementing polymorphism and object I/O. Note that it will suffice to implement the message identifier once, for the input encoding, since the sender of a message knows what output response to expect.

4.4.4 Non-derived classes

An object of a non-derived class is implemented by the circuit shown in figure 17. The interface of the circuit consists of the following signals:

- *Inputs*:
  - CLK and RESET: Clock and reset signals, each one bit wide.
  - SELECT_METHOD: Method identifier for selecting the service to be executed, encoded as defined in equation 48 and 49.
  - IN_PARAMS: Input parameters of the selected service, encoded as defined in equation 42–44.
- *Outputs*:
  - STATE_OUT: Output of the object's current state, to be used for condition synchronisation as described in section 4.5.
  - OUT_PARAMS: Output parameters of the selected service, encoded as

defined in equation 45–47.

- DONE: A handshake signal for signalling request acceptance and completion of a method's execution. The protocol details are described in section 4.5.1.

• *Bidirectional*:

- CHANNELS: Compound signals connecting the object as a client to other objects (servers). The details of channel structure and message exchange are explained in section 4.5.1.

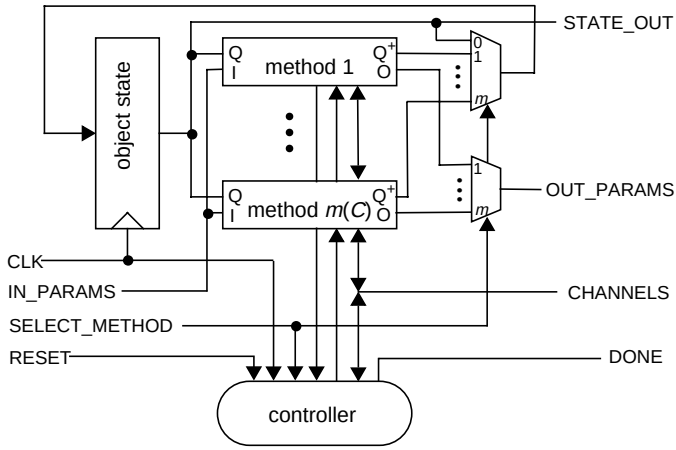


Fig. 17: Implementation of an object of a non-derived class, *C*

A memory element, object state, stores the object's current state. The state is encoded according to equation 32–35. Hence, the memory element must provide $b(C)$ state bits or more. Any surplus bits, caused e.g. by the availability of memory sizes, would remain unused. In figure 17, the memory element is depicted as an edge-triggered register. This shall also stand for any other kind of memory, e.g. SRAM. The use of addressable memory may be preferred over registers for storing (part of) the object's state if the state space is large or organised as an array.

The object's current state and the input parameters, IN_PARAMS, are provided as inputs, Q and I, to a collection of sub-components, each of which contains a data path that implements one of the $m(C)$ methods of the object. The outputs of a sub-component are the method's next state, Q^+, and the output parameters, O. These are computed according to the next state and output functions as defined in equation 10.

As we know from the considerations on synchronisation and concurrency in section 4.2.2, only a single method can be active at a time. This method is selected by the method identifier present at the SELECT_METHOD input. A multiplexer chooses the output computed by the one method under execution and presents it at the OUT_PARAMS output. A second multiplexer selects the next state computed by the active method, which is then fed back into the state register's input. A special situation occurs in the absence of a service request, identified by a 0 at the SELECT_METHOD input. In this case, the unmodified current state is selected as the next state, and the output parameters may take on any, unspecified value.

The controller allows to execute complex methods, e.g. methods with data-dependent loops, in multiple control steps. For this purpose, it has output signals for steering the method's data path, input signals for observing conditions computed in the data path, and internal control states. The controller reads the SELECT_METHOD input so as to know which method to execute. Furthermore, the DONE signal is used for signalling the completion of method execution. The separation of data path and control corresponds to the typical target architecture of high-level synthesis (HLS). We will later aim to delegate data path and controller synthesis to a HLS system.

Both, controller and method data paths have access to the CHANNELS that allow to send messages to server objects. Thereby, a data path can provide values for the input parameters of a service and receive the service's results. The controller, on the other hand, takes care of protocol issues such as waiting for the completion of service execution (see section 4.5.1).

The structure of the object circuit is guided by the model of a finite state machine implementation: There are a memory element for state storage, state transition and output logic, and a feedback of the next state into the state memory. However, the differences should be emphasized: A state transition of the object FSM, defined by the complete execution of a method, can take multiple clock cycles. During this period of time, intermediate state changes can be caused by the sequential execution of attribute value modifications. Note that the data path of the method under execution has continuous access to the object's state and may modify it in any control step via its Q^+ output. It is not the case that the state can be changed only at the end of execution of a method.

Finally, we mention that all object outputs will be registered by HLS. This avoids combinational feedback cycles when objects are interconnected.

4.4.5 Derived classes

Two major alternatives exist for the implementation of inheritance: delegation and expansion. *Delegation* means building the extensions of a derived class around an instance of its direct parent class. *Expansion*, on the other hand, involves a flattening of the inheritance structure at compile time [67]. In the following, both alternatives are considered for implementing an object of a derived class as a digital circuit.

In the delegation implementation of a derived class, *D*, an instance of its direct parent class, *P*, provides the implementation of *P*'s state space and methods. Hence, only the state extension corresponding to *D*'s new attributes and the methods defined in class *D* have to be added. The additional state bits are stored in an extension register that corresponds to q_{ext} of equation 11. The additional methods are implemented by new method sub-components. Furthermore, a controller and multiplexing logic must be created for them.

This outline should suffice for discussing the implications of delegation. Its advantage is that it eases the work of a synthesizer tool because only the new methods and their controller have to be implemented. However, the quality of the resulting circuit is impaired:

- Redundant functionality may be implemented in the separated controllers of the parent instance and the derived class's additions.
- Inherited methods which are never called, e.g. because they have been redefined in the derived class, are implemented by the parent instance.
- A dynamically bound intra-object method invocation of the parent (*self* reference, cf. section 4.2.3) cannot be adapted to invoke a redefined method of the derived class.
- The possibility of sharing resources among methods of *P* and *D* cannot be exploited.
- Multiplexer chains of a length proportional to the depth of the inheritance hierarchy are created and may contribute to the critical path.

We therefore exclude delegation from further considerations and focus on the expansion technique, which facilitates optimizations across class boundaries by flattening of the inheritance hierarchy. This approach has already been used in the FSM definition in section 4.1.5, where the inherited methods have been considered explicitly in the FSM of a derived class.

The structure of an expanded implementation of a derived class is the same as the one that has been devised for a non-derived class in section 4.4.4. It is shown in figure 18, where the parts printed in grey are new with respect to the parent class' implementation.

The interface consists of the inputs, CLK, RESET, SELECT_METHOD and IN_PARAMS, the outputs, STATE_OUT, OUT_PARAMS, and DONE, and the bidirectional CHANNELS for communication with server objects. The complete object state, encoded according to equation 36, is stored in a single memory element. All method data paths, including the inherited ones, are implemented as sub-components at the same level. Their next state outputs are multiplexed by a single, large multiplexer. The same holds for the parameter outputs. A single controller takes control over the execution of all methods.

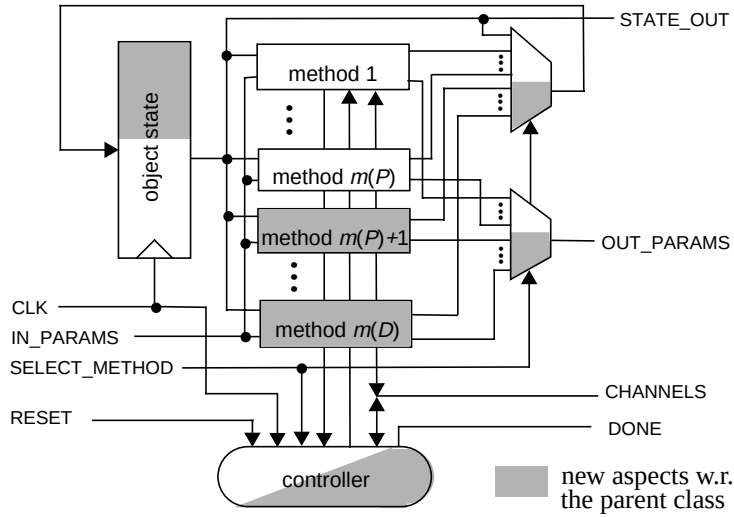


Fig. 18: Implementation of derived object by expansion

By implementing the inherited state and functionality together with the additional attributes and methods of the derived class, the problems of delegation are avoided:

- The single controller can be optimized globally.
- Inherited methods that are never requested can be omitted.
- The inherited methods can share resources with the new ones.
- Multiplexer chains are not created; instead, a tree structure can be used for implementing the large multiplexers in figure 18.

This comes at the cost of increased synthesis complexity compared to delegation.

4.4.6 Implementation of methods

We now investigate the implementation of the method sub-components. A method's functionality is defined by a sequential algorithm which can be synthesized using high-level synthesis techniques. An outline of HLS has been presented in section 2.3.1. Since we will aim to delegate method implementation to an HLS tool, only some basic considerations regarding the structure of the object model and its implementation are made here. These include the method's access to attributes, resource sharing among methods, and the implementation of intra-object method invocation.

The method's algorithm has read and write access to an object's attributes. In section 4.4.1 and 4.4.2, the encoding of the object's state has been chosen so that each attribute is encoded with an individual set of state bits. These are located at a statically known position within the complete state vector. Hence, attribute access is easily implemented by selecting just these known individual state bits.

Resource sharing among methods is enabled by the mutual exclusion of method execution within an object. Hence, a resource can be used in all the different methods of an object without the danger of resource conflicts. Further potential for resource sharing comes from mutually exclusive sections, e.g. branches of a conditional statement, within a method. Respective resource sharing techniques are not further developed in this work because the VHDL code generation to be described in chapter 7 allows to delegate this task to an HLS tool.

To illustrate attribute access and resource sharing, the example of buffer objects (cf. section 4.1.9) is continued. We implement a *LIFO* object as a bounded stack whose *index* attribute designates the next free item above the top of the stack. It is assumed that a service is performed only if its guard expression (cf. section 4.2.6) is true. Hence, *put* and *get* do not need to consider the error cases of a full or empty buffer, respectively. The *put* method stores the value that is to be inserted into the buffer in the item designated by the *index* attribute, and increments *index* thereafter. The *get* method first decrements *index* and then returns the designated item. Using intuitive pseudocode, this reads as:

```
put( val : in Item_t ) {
    item[index] ← val ;
    index ← index + 1 ;
}
```

```
get( val : out Item_t ) {
    index ← index – 1 ;
    val ← item[index] ;
}
```

Data paths that implement these methods are displayed in figure 19. The ASSIGN and SELECT blocks implement the reading and assignment of an array element, respectively. The object's current state, Q, is split into the attributes, INDEX and the ITEM[] array. These are read and partly modified by the data paths, and then joined again into the object's next state, Q^+.

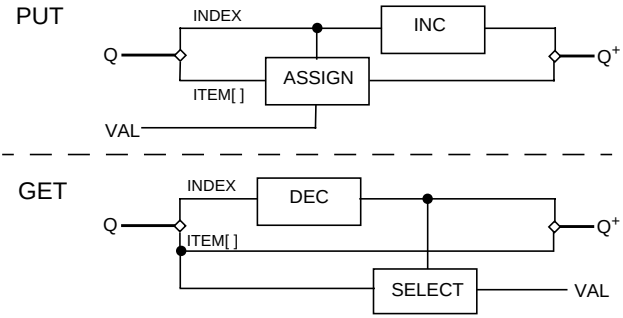


Fig. 19: Datapaths for *put* and *get* methods of *LIFO*

Since *put* and *get* are executed mutually exclusive, their implementations can share resources. For instance, an INC/DEC module can be used instead of the separate incrementer and decrementer, which leads to the circuit shown in figure 20. The individual method sub-components are dissolved after joining them to form a common data path. Additional control inputs have to be provided to switch between the modes of the data path (here: put vs. get) and of its components (here: increment vs. decrement).

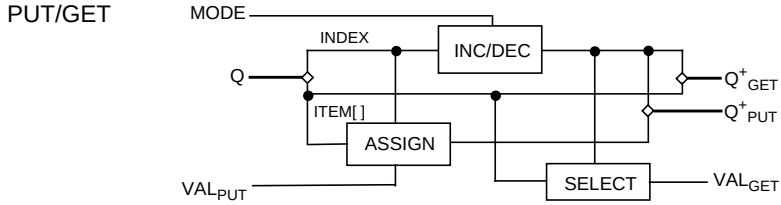


Fig. 20: Datapath implementing *put* and *get* with a shared resource

Finally, intra-object method invocation shall be considered. As described in section 4.2.3, a method of an object can invoke a method of the same object or of an exclusively owned sub-object. In the following, both cases can be regarded as similar since sub-objects are defined and encoded as a part

of their owners' state space (cf. section 4.1.7 and 4.4.1). We implement intra-object method invocation as a part of the invoking method. Note, it is *not* implemented by some communication between method sub-components. Without optimization, this incurs the overhead of multiple implementation of the invoked method: as part of the data path of the invoking method and in its own method sub-component. However, resource sharing can merge these implementations by using the same resources for all of them.

4.4.7 Polymorphic objects

A polymorphic object has been characterised in a way similar to objects with fixed class membership. There were differences only in the structure and encoding of the state space—a polymorphic object has a tag for identifying its class membership—and in the dynamic binding of methods which are redefined in derived classes. In this section, the implementation of these features is described for a polymorphic object with root class *R*.

The basic circuit structure is the same as for a non-polymorphic object of class *R*. That is, it follows figure 17 if *R* is non-derived and figure 18 if *R* is derived. Moreover, the polymorphic object has the same method identifiers, input and output spaces, and the corresponding encodings (cf. section 4.4.3). Only the state space is encoded differently, as described in section 4.4.2.

Dynamic binding has been defined as the invocation of a method's latest redefinition with respect to the current class membership of a polymorphic object. While for each class the latest redefinition of a method can be determined statically, the object's class membership, identified by a tag, changes during system operation. An implementation of dynamic binding therefore has to select the next state and output according to the tag value as shown in figure 21.

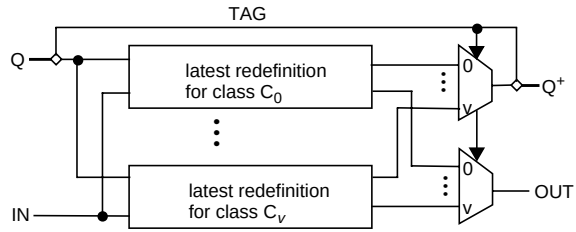


Fig. 21: Implementation of a dynamically bound method

The duality of messages and polymorphic objects is reflected in the circuit structure. While at the top level of object implementation a service is selected according to the method identifier of an incoming message, dynamic binding employs an analogous structure for selecting the implementation of a service depending on a polymorphic object's tag.

Note that the previous section's considerations on method implementation apply to dynamically bound methods. The latest redefinitions of a method are mutually exclusive with respect to each other. Furthermore, they are mutually exclusive with respect to all other methods of an object.

4.5 Implementation of inter-object communication

Having explained the structure of single objects, communication between objects shall now be investigated. The first points to be addressed are the implementation of channels and the description of the message exchange protocol that is executed on these channels. Next, the arbitration of multiple concurrent clients of a server object is explained. As a part of an arbiter circuit, a scheduler that selects a client whose request is granted must be implemented. This leads to the discussion of applicable scheduling policies.

4.5.1 Channels and message passing

A channel connects one client object to one server object. It allows the transmission of the following data:

- The method identifier of the requested service, $m \in M_C$, encoded according to equation 49. The channel contains a signal, REQ, for transmitting this service request. It is connected to the SELECT_METHOD input of the server.
- The input parameters of the service, $inp \in S(in_C)$, encoded according to equation 43. Their value is transmitted over the P_IN signal of the channel, which is connected to the IN_PARAMS input of the server object.
- The output parameters of the service, $outp \in S(out_C)$, encoded as described by equation 46. Their value is transmitted back from the server's OUT_PARAMS output to the client via the channel's P_OUT signal.
- A one-bit handshake signal, DONE, that allows the server to acknowledge message reception and notify the client of the completion of service execution. This signal is connected to the server's DONE output.

Note that, whereas the channel is classified as bidirectional (cf. figure 17 and 18), the signals of which it is composed are unidirectional. Some of these signals lead from the client to the server, the others have the reverse direction. Hence, bidirectionality is not to be understood in the sense of an implementation with tri-stated signals.

The message exchange protocol performed via the channel is displayed in the timing diagram of figure 22. The diagram shows the waveforms of the channel signals. Since a synchronous, clocked implementation is assumed, these waveforms are shown in relation to a global clock signal which is not part of the channel.

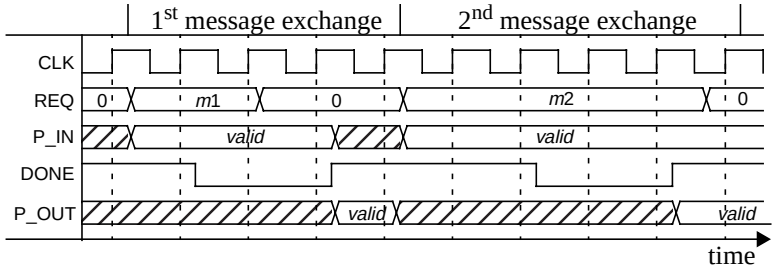


Fig. 22: Message exchange protocol

A message passing operation is started by the client object by assigning the identifier of the requested service to the channel's REQ signal. In the same clock cycle, the input parameters, if any, must be assigned to P_IN, too. The server responds by setting its DONE signal low as soon as it has accepted the request. This takes at least one clock cycle, but may take longer as shown in the second message exchange. After request acceptance, the client may set the REQ signal back to zero. However, it must keep the input parameters valid until service execution is completed. This is signalled by the server by setting the DONE signal high. In the same clock cycle, the server assigns the output parameters of the service to the P_OUT signal, where they remain valid until the next message exchange begins. This leaves the client only one clock cycle to read these output parameters. Note that, even if the client under consideration initiates no new message exchange, this can be done by another, concurrent client.

As soon as one message exchange is finished, i.e., one cycle after the server has risen the DONE signal, the next message exchange may start. Note that this requires the client to set the REQ signal back to zero (no request) in time, i.e., at the latest in the cycle in which DONE is set high (see second

message exchange). Otherwise, the server would assume that the same service is requested again.

Only a single client of a server has been taken into account in the above considerations. In the following section, we extend message passing to consider multiple clients.

4.5.2 Arbiter

The arbiter acts as an agent that allows multiple channels to connect to a server object, which has only a single communication interface. To this end, the arbiter provides to each of the clients an exclusive server interface. With respect to the actual server, the arbiter behaves like a single client.

If, at some time, only a single client has a communication request, the arbiter simply connects this client with the server. If multiple clients issue a request at the same time, the arbiter chooses one of these clients and puts it through to the server. While doing so, the arbiter respects the synchronisation conditions specified by the guard expressions of the server's methods. That is, it accepts a message only if the server is ready to execute the requested service. This is the case if the service's guard expression is true. Otherwise, the respective client is put on hold until its request becomes acceptable.

Figure 23 illustrates the arbitration of three concurrent clients, two producers and one consumer, of a buffer object. It is assumed that the producers issue *put* requests while the consumer requests the *get* service at the same time. The arbiter chooses one of these clients, in this case the second producer, and establishes its connection with the server, i.e., the buffer object. When the service is completed, the arbiter can grant one of the other requests.

An implementation of an arbiter for *N* clients is shown in figure 24. Its interface corresponds to the channel signals: The *N* request, input parameter, output parameter, and handshake (DONE) lines on the left hand allow to connect channels from *N* clients. The signals S_REQ, S_P_IN, S_P_OUT, and S_DONE on the right hand provide the interface to a server. In addition, the server's state is required as an input.

The state, the requests, and their input parameters enable the computation of guard expressions in the Guards sub-component. Its output, the *N* bit wide RDY signal, tells for each of the *N* clients whether or not the synchronisation conditions allow to grant its request, if any. This result is passed on to the scheduler sub-component. The scheduler decides which of the acceptable requests is granted. This decision may require knowledge of the requests if,

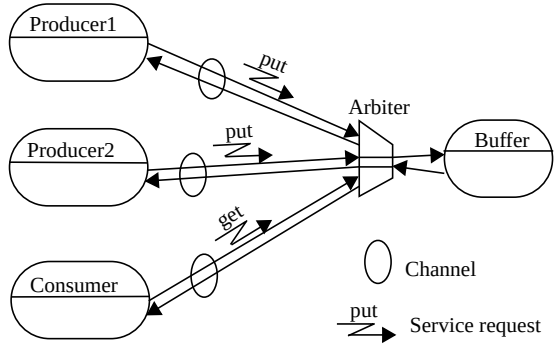


Fig. 23: Communication of concurrent objects

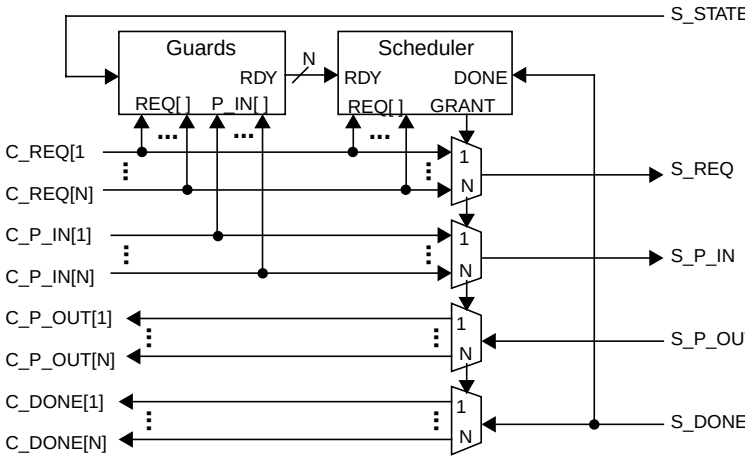


Fig. 24: Arbiter circuit for *N* clients and one server

for example, priorities are associated with requests. Hence, the requests are further inputs to the scheduler. Another input is the S_DONE signal from the server, which lets the scheduler know when the server has completed service execution so that the next request can be accepted.

The scheduler's output, GRANT, is used for controlling the multiplexing and demultiplexing of the channel signals: The granted request and its input parameter are put on the S_REQ and S_P_IN outputs and are thereby sent to the server. The service's output parameters, S_P_OUT, and the server's hand-

shake signal, S_DONE, are demultiplexed to the clients. This is trivial for the output parameters, which can be presented to all clients. However, the DONE signal must be forwarded only to the client whose request has been accepted. The other clients must be provided a stable value of C_DONE = 1. Otherwise, they would all assume their requested service had been executed.

Arbitration is simplified significantly if the guard expressions do not depend on the services' input parameters. This allows to implement the guards sub-component as a part of the server, where it has direct access to the state. Instead of S_STATE, the RDY signal is routed from the server to the arbiter. This signal is now *M* bit wide, where $M = |M_{Server}|$ is the number of the server's services. The *i*-th bit is high if the service addressed by method identifier *i* can be accepted, i.e., its guard expression is true, and low otherwise.

4.5.3 Scheduler

As described in the previous section, the basic functionality of the scheduler is to determine the client whose request is granted. Scheduler functionality is detailed with the following criteria in mind:

- *Implementability*—The scheduler is to be implemented as a digital circuit with a reasonable, small amount of resources.
- *Speed*—The scheduler shall examine all requests in a single clock cycle. When the server is ready to accept a request, this request shall be granted in the same cycle so that its execution can start in the next cycle.
- *Correctness*—The scheduler never accepts a request when it cannot be executed by the server.
- *Fairness*—A client shall not have to wait unnecessarily long for the acceptance of its request.

The notion of fairness [55] has to be detailed further. It can be understood as the absence of starvation; that is, a client that has a request must be served after a finite waiting time. Alternatively, one may demand that requests be served in the order they have been issued. However, ensuring fairness is complicated by the need to take condition synchronisation into account. If a service's guard expression is false, a request of this service must not be accepted, and this should not be regarded a violation of fairness. It may, however, be understood as unfair if the acceptance of other requests prevents the guard of a requested service from becoming true. Hence, we cannot rule out temporarily unacceptable requests from fairness considerations.

For instance, assume that a buffer has run full. Its arbiter may accept *put* and *get* requests alternately. This may be fair for the clients that issue these requests. However, assume there is a third client which wants to put two items at a time. For this client, scheduling is unfair because its request never becomes acceptable. It would be necessary to accept two successive *get* requests to give the third client a chance.

The example emphasizes that fairness can strongly depend on the object's state, the functionality of the requested services, and the order in which requests are accepted. Different from scheduling of independent processes performed by operating systems [156], the interdependence of object services complicates fairness considerations. It is therefore not possible to implement a single scheduler that ensures fairness by construction nor to assess fairness of a scheduling strategy statically. Instead, we have to provide a couple of scheduling policies from which users can choose according to their judgement of suitability for the intended dynamics of an object-oriented model.

4.5.4 Scheduling policies

Scheduling policies for VLSI hardware implementation have been analysed in [62]. They have been classified according to the use of an equal-priority, unequal-priority, rotating-priority, random-delay, or queuing protocol. The assignment of priorities to tasks allows to guide scheduling. For instance, an important service can be assigned statically a high priority. Or, a service request which has already been waiting for a long time can be assigned a higher priority during operation.

In this work, a similar classification is used; see figure 25. We distinguish static and dynamic priority. Equal priority (or, no priority) and unequal priority are among the static priority schemes. Dynamic priority protocols include rotating priority, queuing and externally changed priority.

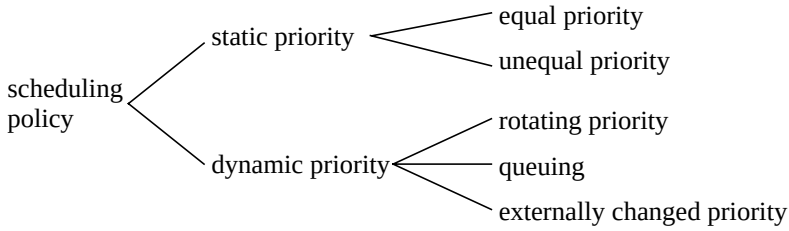


Fig. 25: Classification of scheduling policies

An equal priority protocol involves the non-deterministic choice between simultaneous requests. This can be implemented in hardware with random generators, e.g. linear feedback shift registers. The unequal priority protocol gives each service or client a fixed priority level. A request of a high priority service or from a high priority client is granted before a request of lower priority. If several services or clients share the same priority, a secondary criterion must be applied such as random choice or one of the dynamic policies.

Rotating priority, or round-robin scheduling, shifts the highest priority from one service to the next so that in a period of time each service has a slot in which it has the highest priority. To implement a queuing (FIFO) protocol, the highest priority is assigned to the oldest request. This requires to remember the order in which requests are issued. Finally, while priorities are managed within the scheduler for round robin and queuing protocols, also a protocol with externally changed priorities can be implemented.

4.6 Summary

In this chapter, we have developed a meta-model of the static aspects of objects and their communication following UML constructs. With regard to the parallel nature of hardware, emphasis has been given to concurrency aspects. The concept of guarded methods has been adopted to enable a description style of the condition synchronisation of methods that avoids the problem of inheritance anomaly. A value-based notion of polymorphism and dynamic binding feasible for a hardware implementation has been outlined.

In order to implement objects, their state space encoding has been defined, covering objects of non-derived classes, of derived classes, and polymorphic ones. Likewise, an encoding of an object's input and output spaces has been developed. Circuits for non-derived, derived, and polymorphic objects have been presented. Finally, we have addressed the implementation of communication channels between objects and the arbitration of concurrent requests. The implementation not only ensures mutual exclusion synchronisation, but also respects the synchronisation conditions specified by the methods' guard expressions.

The practical description of an object system, covering its dynamics and functional aspects, can be achieved using an object-oriented language. This will be described in the following chapter. Properties captured by the meta-model can be extracted from the language-based model and used as input for object system synthesis.

Chapter 5

Specifying Object-Oriented Synthesis Models in Objective VHDL

This chapter explains the use of an object-oriented hardware description language, Objective VHDL, for the purpose of specifying object-oriented synthesis models. After giving specific reasons for the choice of Objective VHDL, we show how non-derived and derived classes and their functionality are implemented with this language. Another section is devoted to the declaration and use of objects. This involves a treatment of polymorphism and message passing. Thereafter, concurrency and proposed mechanisms for modelling condition synchronisation and request arbitration are discussed. Finally, we present the integration of object synthesis into the architecture of tools available for this language.

5.1 Language choice

The choice of an object-oriented language for hardware design should be a matter of serious consideration since the language and its expressive power impact not only the language's ease-of-use, but also tool development. In this work, a variant of the hardware description language VHDL has been preferred over the use of an object-oriented programming language.

The main reason for this decision is that the hardware meta-model of object-orientation presented in the previous chapter is based on a notion of static objects and channels to represent static hardware resources and their fixed interconnection. VHDL provides concepts such as signals, generate statements, generics, and the elaboration of models, that enable static modelling. These can be used in combination with new object-oriented features so as to facilitate the modelling of static object systems. On the other hand,

OOP languages, in particular C++ and Java, rely a lot on dynamic concepts such as run-time object allocation and pointers or references. This makes the extraction of static properties harder. Moreover, it requires to impose restrictions on the use of these languages. This affects not only the available modelling power, but also the language's ease-of-use as designers have to pay attention to the limitations.

We believe that the addition of object-oriented features to VHDL can contribute to more abstract modelling of hardware objects and their communication. Their combination with VHDL's concurrency features is inevitable for modelling parallel hardware. Furthermore, VHDL's structure modelling features, that include entities, architectures, components, and their configurations, are valuable for organising a complex system.

The synthesis of models that use the object-oriented extensions together with the synthesizable subset of VHDL enables an automatic transition to implementations at lower abstraction levels. These implementations can again be expressed in object-oriented VHDL or its VHDL subset. This eases not only the synthesis task, but also the joint simulation of models at different description levels.

The choice of an object-oriented variant of VHDL is another issue. In this work, Objective VHDL has been chosen for the demonstration of concepts. The language design of Objective VHDL is not a subject of this thesis, nor are modelling issues. It is not claimed that Objective VHDL as a language is superior over other OO-VHDL dialects. However, we show that and how Objective VHDL can be applied for describing object systems.

Our language choice is primarily based on the availability of an Objective VHDL compiler front-end that has been developed as an extension of a commercial VHDL product. Likewise, a prototype translator that creates VHDL for simulation purposes is available. This degree and quality of tool support makes Objective VHDL favourable for the purpose of demonstrating the hardware synthesis of object-oriented descriptions. No other OO-VHDL dialect is currently supported to this extent by tools, if any.

The Objective VHDL language and the VHDL translator have been developed with the author's contribution in the ESPRIT project REQUEST. The language is defined in the same style as VHDL by an extension [101] to VHDL's language reference manual [74]. The tools are available for various Unix operating systems and Windows 95/98/NT [127]. In the following, the relevant portion of Objective VHDL and its tool architecture are outlined. This provides the foundation for demonstrating the synthesis concepts of chapter 4 in the remainder of this work.

5.2 Declaration of classes

Objective VHDL facilitates object-oriented modelling by adding class types to VHDL's type system. This is achieved by extending VHDL's grammar so that class types can be defined in addition to scalar, composite, access, and file types:

```
type_definition ::=
    scalar_type_definition
    composite_type_definition
    access_type_definition
    file_type_definition
    class_type_definition          -- new
```

A class type definition can be the declaration of a non-derived class type, of a derived class type, or of a class type body:

```
class_type_definition ::=
      class_type_declaration
    | derived_class_type_declaration
    | class_type_body
```

These new language features of Objective VHDL are explained in the following sections.

5.2.1 Class type declarations

A non-derived class is declared with Objective VHDL's class type declaration construct, where the optional keyword **abstract** can be used to declare the class as an abstract one:

```
class_type_declaration ::=
    [ abstract ] class
        [ formal_generic_clause ]
        { class_type_declarative_item }
    end class [ class_type_simple_name ]
```

This syntax is to be used in the context of a type declaration. Hence, the declaration of a non-abstract class type, including the optional repetition of the class type's name, effectively reads as follows:

```
type name is class ... end class name;
```

The optional generic clause allows to declare generics as known from VHDL entities. Thereby, a class can be parameterized with generic values. The use of generic types in the sense of section 4.1.8 is currently not supported by the Objective VHDL language. Likewise, the language provides no interface inheritance. These design decisions have been taken in order to limit the complexity added to VHDL. Abstract classes can be used in most cases to achieve the desired effects of interface specification and type genericity.

An arbitrary number of the following declarative items can be included in the class type declaration:

```
class_type_declarative_item ::=
      class_attribute_declaration
    | class_type_common_declarative_item
    | class_type_object_configuration
```

The class attribute declaration facilitates the declaration of attributes in the object-oriented sense (cf. section 4.1.2). These are told apart from VHDL's attributes by using the keyword combination **class attribute**. In the following, we speak of class attributes and attributes interchangeably and use the term VHDL attribute if meant in the VHDL sense.

```
class_attribute_declaration ::=
    class attribute identifier : subtype_indication [ := expression ] ;
```

The identifier is the name with which a class attribute can be referenced. The subtype indication specifies the (sub-)type of the class attribute, and thereby defines its state space. An optional expression allows to assign the attribute's initial value explicitly. If it is omitted, the initial value is defined according to VHDL rules as the leftmost value of the attribute's type.

Further declarative items, common to all instances of a class, are:

```
class_type_common_declarative_item ::=
      type_declaration
    | subtype_declaration
    | constant_declaration
    | subprogram_declaration
    | alias_declaration
    | use_clause
```

This facilitates the declaration of types, subtypes, and constants which are local to the class. Furthermore, it is possible to declare aliases and to import declarations by means of a use clause. Most important for the object model

are subprogram declarations which facilitate the declaration of the class' methods.

A class can be instantiated as a signal, variable, or constant object. A class type object configuration allows to declare items which are *not* common to all of these object kinds:

```
class_type_object_configuration ::=
    for object_specification { , object_specification }
        { class_type_common_declarative_item }
    end for ;

object_specification ::=
    signal | variable | constant
```

The declarations permitted within such an object configuration are the same as above. However, they are only available with the kind of objects specified in the object specification list. For instance, the declaration of a method that modifies the state of an object does not make sense for a constant object because its state cannot be modified. Hence, this method would be declared only for signal and variable objects:

```
for signal, variable
    procedure modify;
end for;
```

The following listing shows the declaration of the abstract class Buffer_t, corresponding to section 4.1.9. The example has been changed with respect to its UML notation by removing the parameterization of stored items. Instead of being of a generic type, the stored items are declared as integer values with parameterized range. This shortens the example so that the Objective VHDL code and generated VHDL code can be presented in this work. A more generic implementation would declare the item type as an abstract class and store polymorphic objects of that root class. By deriving several types of stored items from the abstract root class, the user would be able to use the buffer with all these types. Note that we cannot name the type Buffer since this is a reserved word in VHDL.

```
type Buffer_t is abstract class
    generic(
        size : Positive;          -- Max. no. of entries in buffer
        bits : Positive           -- No. of bits per entry
    );
    subtype Item_t is Integer range 0 to 2**bits – 1;
```

```
    type Buffer_array is array ( 0 to size – 1 ) of Item_t;
    class attribute item : Buffer_array := (others => 0);
    function is_full return Boolean;
    function is_empty return Boolean;
    for variable, signal
        procedure put( val : in Integer );
        procedure get( val : out Integer );
    end for;
end class Buffer_t;
```

The class has two generic values, the maximal number of entries that the buffer can store, size, and the bit-width of each entry, bits. It declares the item type Item_t as the range of non-negative integers that can be represented with the generic number of bits. Next, an array type with size elements of the item type is declared. The array type serves as the type of the class attribute item which will later be used for storing the items. The array elements of this attribute are initialised with zero.

Two boolean functions, is_full and is_empty, correspond to the respective methods of the UML model. Since they do not modify the object's state, they are declared in the common part of the class which makes them applicable to all kinds of objects. The put and get procedures, on the other hand, perform a state modification and are therefore declared only for signal and variable objects. Note that get cannot be implemented as a function with an integer return value because it removes an item from the buffer, whereas a function in VHDL and Objective VHDL, even if it is declared as impure, must not perform any state modifications.

5.2.2 Derived class type declarations

The derived class type declaration of Objective VHDL provides the following syntax for deriving a new class, optionally abstract, from an existing parent class, and for augmenting it with additional declarative items:

```
derived_class_type_declaration ::=
    new [ abstract ] class class_type_name with
        [ formal_generic_clause ]
        { class_type_declarative_item }
    end class [ class_type_simple_name ]
```

In the context of a type declaration, a class D is derived from a parent class P as follows:

```
type D is new class P with ... end class D;
```

Again, the repetition of the class identifier at the end of the declaration is optional. The parent class can be any class, abstract or non-abstract, derived or non-derived, that is analysed before and visible to the derived class. This ensures that inheritance is free of cycles by preventing the derived class from also being a parent of its parent class.

Any generics declared in a derived class exist in addition to the generics, if any, of the parent class. Objective VHDL employs the notion of an effective generic list for defining inheritance of generics. The effective generic list of a derived class is the concatenation of the parent's effective generic list and the generics declared in the derived class. The effective generic list of a non-derived class comprises only the class' own generics.

The declarative items allowed in the derived class are the same as for a non-derived class: class attributes, types, subtypes, constants, subprogram declarations, aliases, and use clauses. As well, a derived class can contain object configurations that allow to limit their contained declarations to signal, variable, or constant objects, or a combination thereof. All these declarations are additional to the declarations of the parent class.

An attribute declaration in a derived class therefore extends the inherited state space. This holds even if the new attribute is declared with the same name as an inherited one. In this case, both attributes co-exist while the new attribute hides the inherited one so that in the derived class any use of the name refers to the new attribute. Methods inherited from the parent class, however, still reference the now hidden attribute.

A similar mechanism, combined with VHDL subprogram overloading, is applied to methods. Method subprograms are distinguished not only by their name, but also by the types of their parameters and return value, if any. If the name and the parameter and return type profile of a new method declared in a derived class are the same as those of an inherited method, the new method hides the inherited one. It is a redeclaration if in addition the parameter names, storage classes (signal, variable, constant) and modes (in, out, inout) coincide.

The following listing shows the declaration of derived classes, LIFO and FIFO, whose parent class is Buffer_t. LIFO declares and initialises one additional class attribute, index, while FIFO comprises three new attributes, first, nxt, and empty. The meaning of these attributes is explained in section 5.2.4. The subprograms declared in both classes are all redeclarations of the respective methods of the parent class.

```
type LIFO is new class Buffer_t with
    class attribute index : Natural range 0 to size := 0;
    function is_full return Boolean;
    function is_empty return Boolean;
    for variable, signal
        procedure put( val : in Integer );
        procedure get( val : out Integer );
    end for;
end class LIFO;

type FIFO is new class Buffer_t with
    class attribute first : Natural range 0 to size – 1 := 0;
    class attribute nxt : Natural range 0 to size – 1 := 0;
    class attribute empty : Boolean := true;
    function is_full return Boolean;
    function is_empty return Boolean;
    for variable, signal
        procedure put( val : in Integer );
        procedure get( val : out Integer );
    end for;
end class FIFO;
```

5.2.3 Class type body

A class type body provides an implementation of a non-derived or derived class type. It is defined by the following Objective VHDL grammar production:

```
class_type_body ::=
    class body
        { class_body_declarative_item }
    end class body [ class_type_simple_name ]
```

As it appears as part of a type declaration, a complete class body reads:

```
type name is class body ... end class body name;
```

The name must be the same as the name of the class type declaration or derived class type declaration to which the body belongs. It is an error if no such corresponding declaration exists. A class body must be defined for each non-abstract class. An abstract class may, but does not need to have a class body.

Inside the class body, class attributes, common declarative items, and object configurations can be declared:

```
class_body_declarative_item ::=
      class_attribute_declaration
    | class_body_common_declarative_item
    | class_body_object_configuration
```

A class attribute of the class body contributes to an object's state just like an attribute declared in the class type declaration does. However, the former is visible only in the class body itself (private in C++ terms) while the latter can also be read and modified directly by new methods of derived classes (protected). Public attributes, which are accessible everywhere, are not provided by Objective VHDL.

Types, subtypes, constants, subprogram declarations, subprogram bodies, aliases, and use clauses are allowed as common declarative items of the class body. That is, all common declarative items permitted in a (derived) class type declaration may also occur in a class body. In addition, it is possible to declare subprogram bodies, i.e., to implement a subprogram.

```
class_body_common_declarative_item ::=
      type_declaration
    | subtype_declaration
    | constant_declaration
    | subprogram_declaration
    | subprogram_body
    | alias_declaration
    | use_clause
```

Like private class attributes, all these items are only visible in the class body itself. This holds also for subprograms first declared within the class body. Such subprograms are private methods. On the other hand, a subprogram that is declared in a class type declaration is a public method. Protected methods do not exist in Objective VHDL.

The implementation (body) of a method subprogram, be it public or private, is always specified in a class body. It has access to the class' attributes. A procedure method can read and modify an attribute value while a function method can only read, but not modify, the object's state.

A class body object configuration enables the declaration of items that are specific to signal, variable, or constant objects. This includes, in particular, method implementations (subprogram bodies). In a non-abstract class, all methods must have such an implementation. In an abstract class, the implementation is optional.

```
class_body_object_configuration ::=
    for object_specification { , object_specification }
        { class_body_common_declarative_item }
    end for ;
```

In an object configuration for variable, a class attribute can and must be accessed like a variable. In an object configuration for signal, the class attribute appears to be a signal. This does not only imply the use of different assignment operators. The semantics of assignments are different, too: A variable assignment causes an immediate state transition while a signal assignment takes at least one VHDL delta cycle to become effective. Moreover, VHDL signal attributes such as 'EVENT can be applied to a class attribute in a signal object configuration and only there. Finally, the object configuration for constant commits methods to considering the class attributes as constants. These rules reflect the opportunities and restrictions related to class instantiation as a variable, signal, or constant, respectively.

5.2.4 Example

This section demonstrates the use of Objective VHDL for implementing the buffer data types known from previous examples. The organisation of the internal data structures of LIFO and FIFO buffers is shown in figure 26. In order to implement LIFO functionality, the item array is managed as a stack. The class attribute index designates the next free item on top of the stack. In the FIFO class, the item array is organised as a circular buffer. The class attribute first stores the index of the oldest entry while the attribute nxt designates the next free position in the buffer. If first equals nxt, the buffer is either empty or full. This is distinguished with the additional boolean attribute empty.

The implementation of methods must use the class attributes according to the above description and maintain the correct organisation of data structures. This is achieved for class LIFO by declaring its class body as follows:

```
type LIFO is class body

    function is_full return Boolean is
    begin
        return index = size;
    end;

    function is_empty return Boolean is
    begin
```

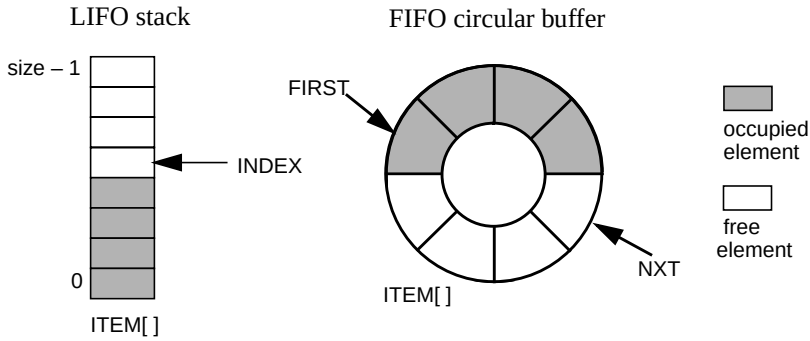


Fig. 26: LIFO and FIFO data structures

```
        return index = 0;
    end;

    --  for signal : see later

    for variable

        procedure put( val : in Integer ) is
        begin
            assert not is_full report "LIFO overflow" severity failure;
            item( index ) := val;
            index := index + 1;
        end;

        procedure get( val : out Integer ) is
        begin
            assert not is_empty report "LIFO underflow" severity failure;
            index := index – 1;
            val := item( index );
        end;

    end for;

end class body LIFO;
```

Note that different implementations of the put and get methods are provided for signal and variable objects. Only the implementation for variables is shown here. Signal objects and their concurrent use are addressed in section 5.4.

The class body of FIFO implements, in addition to the public methods is_full, is_empty, put, and get, a private method, next_index. This method increments an index modulo the buffer size so as to respect the buffer's circular organisation. It can be synthesized even if the buffer size is not a power of two, whereas VHDL's **mod** operator cannot.

```
type FIFO is class body

    function next_index( index : in Integer ) return Integer is
    begin
        -- synthesizable implementation of: return (index + 1) mod size;
        if index + 1 < size then return index + 1;
        else return index + 1 – size;
    end;

    function is_full return Boolean is
    begin
        return nxt = first and not empty;
    end;

    function is_empty return Boolean is
    begin
        return empty;
    end;

    -- for signal : see later

    for variable
        procedure put( val : in Integer ) is
        begin
            assert not is_full report "FIFO overflow" severity failure;
            item( nxt ) := val;
            nxt := next_index( nxt );
            empty := false;
        end;
        procedure get( val : out Integer ) is
        begin
            assert not is_empty report "FIFO underfl." severity failure;
            val := item( first );
            first := next_index( first );
            empty := first = nxt;
        end;
    end for;

end class body FIFO;
```

No class body is declared for class Buffer. This is legal as Buffer is abstract. A body declaration of an abstract class makes sense only if some functionality can be factored out as it is to be implemented the same way for the derived classes. This is not the case in this example.

5.3 Declaration and use of objects

Once a class is defined, it can be used with the VHDL mechanisms that are part of Objective VHDL. This includes the declaration of class-typed signals, variables, and constants, by which objects are instantiated. So-called class-wide types facilitate the declaration of polymorphic objects. These aspects are addressed subsequently in this section.

5.3.1 Instantiation of classes

A non-abstract class type or non-abstract derived class type can be used like every VHDL type. This makes it possible to declare VHDL objects, i.e. signals, variables, and constants, with a class type. Such VHDL objects are considered as objects in the object-oriented sense. Their value represents the object's state.

Note that shared variables can be declared with a class type, too. In the future, however, a re-standardisation of VHDL will likely include the shared variables language change specification [165] as a part of the standard. This will restrict shared variables to so-called protected types. Since shared variables are not synthesizable according to the draft VHDL synthesis standard [75] and not supported by most synthesis and simulation tools, we exclude them from further considerations.

It is also possible to declare an array of a non-abstract class type or a record with non-abstract class-typed elements. The declaration of a signal, variable, or constant with such a composite type yields a respective number of objects.

Moreover, Objective VHDL permits the declaration of class access types and the dynamic allocation of objects. Likewise, it is possible to declare a file type of a class type and to use it in the declaration of file objects. Both, access and file types, are not synthesizable and therefore excluded from further considerations.

The previous instantiation mechanisms do not require any VHDL syntax change. Only for the instantiation of generic classes, i.e., class types which

have a non-empty effective generic list, an addition to the VHDL grammar is required:

```
subtype_indication ::=
      [ resolution_function_name ] type_mark [ constraint ]          -- VHDL
    | [ resolution_function_name ] generic_class_type_mark
                                   generic_map_aspect                -- new
```

This allows to provide actual values for the generics of a generic class type when the class type is used in some other declaration. For instance, a variable LIFO object with eight entries of three bits each can be declared as:

```
variable lifo_object : LIFO generic map( size => 8, bits => 3 );
```

This is only a shortcut as a class type with a generic map is defined as an anonymous constrained subtype of the more general unconstrained class type without generic map. An explicit declaration using a named subtype can be made as follows:

```
subtype LIFO_8_3 is LIFO generic map( size => 8, bits => 3 );
variable lifo_object : LIFO_8_3;
```

5.3.2 Declaration of polymorphic objects

The declaration of polymorphic objects is enabled by Objective VHDL's predefined CLASS attribute, which is an attribute in the VHDL sense. If this attribute is applied to a class type T, it yields the so-called class-wide type of T which comprises T and all classes derived from T. This is defined by the following addition to VHDL's predefined attributes [74]:

Predefined attribute: T'CLASS

Kind:	Type
Prefix:	class type T
Result type:	type definition
Result:	the class-wide type comprising T and its derived classes

A signal or variable declared with the class-wide type T'CLASS is a polymorphic object with root class T. The same holds for an array element or record element of class-wide type. Note that it is possible to declare a class-wide constant, too.

The class-wide (polymorphic) object is uninitialized until it is assigned a value, i.e. a state and class membership. For a constant object, an assignment can be made only at the place of its declaration. This determines its unmodi-

fiable state and class membership. The state and class membership of a variable or signal of class-wide type, on the other hand, can change during system operation. This makes class-wide signals and variables polymorphic objects.

A polymorphic object can be declared with an abstract root class. In this case, the object cannot take on the state and class membership of the root class, but of any non-abstract class derived from the root class. Likewise, a polymorphic object can have a generic root class or a root class from which generic classes are derived. Since all type instances of a generic type are constrained subtypes of the generic type, the class-wide type comprises them all.

The change of a polymorphic object's class membership is caused by assignments to the object. This topic is discussed in the next section.

5.3.3 Assignment

Assignment comprises the assignment of an initial value to a signal, variable, or constant at the place of its declaration, the variable assignment ':=', and the signal assignment '<=' of VHDL. An assignment involves a target and an expression which yields the value to be assigned to the target. This section deals with assignments whose target is an object.

We first consider *non-polymorphic* target objects. Such an object can be assigned a value of its class type, and only of its class type. If the assigned expression is of a class type, this can be checked by an analyser: If expression and target type are the same, the assignment is legal since the types are compatible. Otherwise, the assignment is rejected since the types are incompatible.

If the assigned expression is of a class-wide type, its class membership is not known during analysis. However, its root class is known. If the target object's class is the same as or derived from the expression's root class, types are potentially compatible since the current class membership of the assigned expression may be, but does not necessarily need to be the same as the target's class. Otherwise, the assignment is rejected as types are incompatible. A potentially compatible assignment is accepted by the analyser, but can lead to a run-time error when the expression's type does not match the target type.

The above type considerations are the same if the target object's type is an instance of a generic class type. As in VHDL, analysis considers only the types. A mismatch of subtype constraints, that is, a mismatch of the actual generic values of the target and expression subtypes, is detected at run time.

When analysing an assignment that involves a *polymorphic* target object, we take into account that this object can take on the class membership of its

root class and all classes derived from it. Hence, an assignment of an expression whose type is one of these classes is compatible. The same holds if the expression is of a class-wide type whose root class is the same as or derived from the target's root class. If, however, the expression's root class is a parent of the target object's root class, the expression's current class membership may be, but is not necessarily acceptable for the target object. Hence, such an assignment is potentially compatible only. In all other cases, the assignment is incompatible.

Assignments with incompatible types must be rejected by a language analyser. Potentially compatible assignments pass analysis, but may result in run-time failure. Compatible assignments are always legal from the type perspective. Only if generic class types are involved, a run-time subtype error may occur. A digital circuit implementation ignores such errors.

5.3.4 Tags

In accordance with the considerations of section 4.3.2, Objective VHDL manages for each polymorphic object and for each class-wide expression a tag that determines its class membership. Users do not have to care for the introduction and updating of this information. They can, however, obtain the tag value that is associated with a class type. For this purpose, the VHDL attribute TAG is pre-defined in Objective VHDL:

Predefined attribute: T'TAG

Kind:	Value
Prefix:	class type T
Result type:	*universal_tag*
Result:	the tag (type identifier) that corresponds to class type T

The TAG attribute can be applied to an object, too. If the object is polymorphic, i.e. declared with a class-wide type, this yields the tag that represents its current class membership. For a non-polymorphic object declared with a class type, the class type's tag is returned:

Predefined attribute: S'TAG, V'TAG, C'TAG

Kind:	Function
Prefix:	signal S, variable V, or constant C of a class type or class-wide type
Result type:	*universal_tag*
Result:	if S, V, or C is class-wide: current value of the object's type tag

if S, V, or C is of a (derived) class type: the type tag
that corresponds to the (derived) class type

The tag values have a type, *universal_tag*, that is predefined in Objective VHDL. The following relational operators are predefined, too, allowing to compare tags:

```
function "=" ( anonymous, anonymous : universal_tag ) return Boolean;
-- A = B is true if A and B are the same tags; otherwise: false.
function "/=" ( anonymous, anonymous : universal_tag ) return Boolean;
-- A /= B is defined as not( A = B ).
function "<" ( anonymous, anonymous : universal_tag ) return Boolean;
-- A < B is true if class B is derived from A and A /= B; otherwise: false.
function ">" ( anonymous, anonymous : universal_tag ) return Boolean;
-- A > B (A is parent of B) is defined as B < A.
function "<=" ( anonymous, anonymous : universal_tag ) return Boolean;
-- A <= B is defined as A < B or A = B.
function ">=" ( anonymous, anonymous : universal_tag ) return Boolean;
-- A >= B is defined as A > B or A = B.
```

Note that these relational operators represent the partial order defined by the inheritance tree, not a total order. Hence, **not**(A < B) does *not* imply A >= B since A and B may be unrelated by inheritance.

Users can apply tag comparison, for instance, in order to determine at run-time whether or not a potentially compatible assignment can be executed. This allows to invoke user-defined error handling code instead of causing a simulation failure.

Objective VHDL does not permit the user-defined modification of tags. The only way to change the tag value of a polymorphic object is by assigning a state of another class to the object. In the course of such an assignment, tag updating is performed automatically. The definition of a tag assignment operation would facilitate the user-defined manipulation of class membership. This would not pose a problem to synthesis, but might impair the integrity of the object's internal state.

5.3.5 Message passing

In Objective VHDL, methods are procedures and functions declared in a class construct. The following extension of VHDL's procedure and function calls enables the invocation of methods with an object, i.e., message passing:

```
procedure_call ::=
      procedure_name [ ( actual_parameter_part ) ]                  -- VHDL
    | prefix . class_procedure_name [ ( actual_parameter_part ) ]   -- new

function_call ::=
      function_name [ ( actual_parameter_part ) ]                   -- VHDL
    | prefix . class_function_name [ ( actual_parameter_part ) ]    -- new
```

The prefix must be an object. As explained in section 5.3.1 and 5.3.2, this includes signals, variables, constants, record elements, and array elements declared with a class type or class-wide type. Inside a class construct, the prefix this is pre-defined and can be used for expressing self-referentiation in the sense of section 4.2.3. Moreover, inside a class, a method of the class can be invoked without prefix by using the VHDL syntax of a procedure or function call. This corresponds to sending a message to the ε reference of section 4.2.3.

Methods declared in an object configuration for signal, variable, or constant can be invoked only with signal, variable, or constant objects, respectively. Methods declared in the common part of the class can be invoked with any kind of object.

If the object designated by the prefix is polymorphic, dynamic binding as defined in section 4.3.4 is applied. The same holds if the prefix this is used.

As messages are being sent to a server object that is identified by a prefix, a channel construct does not exist in Objective VHDL. However, channels can be extracted from an Objective VHDL model: Each external server object[1] that is addressed by means of a prefixed method invocation from within a client object requires a channel from the client to the server.

A client gets to know an external server object if the server is passed as a method parameter to the client. Objective VHDL does not allow the client to memorise its server by storing a reference to it in a reference variable. Hence, at each method invocation, all server objects that might be required for executing the method must be passed as a parameter. This has already been identified as a modelling inconvenience in [143][166] for a related OO-VHDL dialect. We therefore suggest an extension of Objective VHDL's class type and derived class type declaration for a port clause:

```
class_type_declaration ::=
    [ abstract ] class
        [ formal_generic_clause ]
```

1. As opposed to an exclusively owned sub-object.

```
        [ formal_port_clause ]     -- suggested extension of Obj. VHDL
        { class_type_declarative_item }
    end class [ class_type_simple_name ]

derived_class_type_declaration ::=
    new [ abstract ] class class_type_name with
        [ formal_generic_clause ]
        [ formal_port_clause ]     -- suggested extension of Obj. VHDL
        { class_type_declarative_item }
    end class [ class_type_simple_name ]
```

This would allow to pass signal objects via ports to a client. The inheritance of ports would be defined in analogy to generics by means of an effective port list. A port map would have to be provided when an object of a port-equipped class is declared. This is enabled by a suggested extension of subtype indications, analogous to their extension for generic maps:

```
subtype_indication ::=
      [ resolution_function_name ] type_mark [ constraint ]          -- VHDL
    | [ resolution_function_name ] class_type_mark
            [ generic_map_aspect ]  -- Objective VHDL
            [ port_map_aspect ]     -- suggested extension to Obj. VHDL
```

The servers used by a client object, and hence the channels from the client to its servers, are

- all objects that are passed as actuals in the port map of the instantiation of a client object and
- all objects that are passed as actuals in the parameter association list of any invocation of a method of the client object.

Note that an association of a port with a server object establishes an unchanging connection of the client with a server. Hence, a port most closely corresponds to the notion of a channel. On the other hand, in different invocations of the same method of a client, different servers may be passed via the same formal parameter. We deal with this situation by considering the formal subprogram parameter as a pseudo object that only connects with the actual object the method has been invoked with. This is channel switching as in section 4.3.3, ensuring the staticness of channels.

5.3.6 Example

The following listing shows the declarations of classes, Producer and Consumer, that use a polymorphic object of root class Buffer_t as a server. While

the Producer class is connected to its server through a port declaration, the consumer receives its server by parameter passing at each invocation of its method consume.

The possession of a channel to a server, be it via a port or a method parameters, enables the classes to request services from this server. In its produce method, the Producer issues a put request to place a token into the buffer. The Consumer, on the other hand, gets a token from the server in its consume method.

Note that these classes may have additional methods to define the token value that is produced and to obtain a token value that has been received.

```
type Producer is class
    port( server : inout Buffer_t'CLASS );
    class attribute token : Integer := 0;
    for variable
        procedure produce;
    end for;
end class Producer;

type Producer is class body
    for variable
        procedure produce is
        begin
            server.put( token );
        end;
    end for;
end class body Producer;

type Consumer is class
    class attribute token : Integer := 0;
    for variable
        procedure consume( signal server : inout Buffer_t'CLASS );
    end for;
end class Consumer;

type Consumer is class body
    for variable
        procedure consume( signal server : inout Buffer_t'CLASS ) is
        begin
            server.get( token );
        end;
    end for;
end class body Consumer;
```

In the following code excerpt, objects are instantiated and used. One polymorphic buffer object is declared as a signal, buffer_obj. Two producer objects are declared as variables of a process and connected to the buffer by their port map. A third client, a consumer object, is instantiated as another variable. It receives its server by parameter passing. Hence, the object system is characterised as follows:

- Objects: Ω = {server, producer_obj1, producer_obj2, consumer_obj}
- Channels: ζ = { (producer_obj1, server), (producer_obj2, server), (consumer_obj, server) }

```
signal buffer_obj: Buffer_t'CLASS;

process
    variable producer_obj1 : Producer port map( server => buffer_obj );
    variable producer_obj2 : Producer port map( server => buffer_obj );
    variable consumer_obj : Consumer;
begin
    ...
    producer_obj1.produce;
    producer_obj2.produce;
    consumer_obj.consume( buffer_obj );
    ...
end process;
```

5.4 Concurrency

In the previous example, the server object has been used only from within a single sequential process. This section now addresses the description of concurrently used objects in Objective VHDL. The language does not currently provide features to support modelling of concurrent objects. The definition of such features is desirable, but not subject of this work. Instead, we devise a modelling scheme that allows to express the observation of guard expressions and the choice of a scheduling policy for acceptable requests in Objective VHDL as it is.

While the approach of defining a modelling guideline is similar to [145], our technical choices are different. While Schumacher explicitly describes the details of inter-object communication using VHDL signals, this work focuses on a description style that allows to rely on method calls for communication between concurrent objects. Furthermore, this work is unique in enabling the choice of a scheduling strategy for acceptable requests. Most

importantly, the modelling style presented here allows to transform request scheduling and the observation of guard expressions into a hardware implementation.

5.4.1 Modelling of guard expressions

Variables can be declared and used only in sequential parts of a VHDL or Objective VHDL model, whereas signals can be used concurrently. Hence, concurrent objects must be modelled as signals. The observation of guard expressions is therefore to be integrated into the method implementation for signals. For this purpose, the following procedure is pre-defined in the Objective VHDL language environment:

```
procedure guard( expression : Boolean );
```

The procedure shall be called only as the first statement of a method. If the boolean expression is true, the procedure returns control to its calling method. If the expression is false, the procedure quits the execution of the calling method and re-queues the service request which has lead to the method's invocation. This means that the service request is suspended, but not finished. It is considered again for execution when the object's state has changed.

We assume that a method implementation for variable objects exists. Our modelling style delegates the implementation of a service to this implementation by copying the state of the signal object into a variable, executing the implementation with this variable object, and writing the resulting state back to the signal object. This not only saves the effort of re-writing an implementation for variables with signal semantics in mind. It also avoids the shortcomings of signal synthesis present in the high-level synthesis tool that we use as a back-end.

The template of a method implementation for signal objects is shown in the following listing. The method first checks whether the guard expression is fulfilled. If this is not the case, the guard procedure quits the method's execution and re-queues its execution request. Otherwise, the method proceeds immediately. It copies the object's state, designated by the predefined symbol this, into a variable, object_copy, of the object's own class type. The method implementation for variable is executed with the copy and the parameter values that have been passed to its signal implementation. Finally, the modified state of the variable is copied back to the signal object.

```
for variable, signal
    procedure method( <parameters> );
end for;
...
for signal

    procedure method( <parameters> ) is
        variable object_copy : <class_type>;
    begin
        guard( <guard_expression> );
        object_copy := this;
        object_copy.method( <parameters> );
        this <= object_copy;
    end;

end for;
```

It should be emphasized that the guard procedure used in the above template cannot be implemented in Objective VHDL nor VHDL. Instead, it has to be implemented by the run-time system of an Objective VHDL simulator. Likewise, it has to be recognised and transformed into a hardware structure by the Objective VHDL synthesis tool. Synthesis, as it will be presented in chapter 7, can avoid the copy and copy-back operations. Hence, the templates do not incur hardware overhead in terms of redundant state bits or unnecessary delay for data transfer.

A class-typed signal does not have to have a VHDL resolution function if only guarded methods are invoked with it from multiple processes. The guard mechanism replaces the VHDL resolution mechanism.

5.4.2 Scheduling policy

To enable the user-defined choice of a scheduling policy and even the user-defined implementation of schedulers, a library named SCHEDULERS is defined. This library contains the following interface of a scheduler entity:

```
entity Scheduler is
    generic(
        no_clients : Positive;
        req_bits : Positive );
    port(
        clk : in Std_logic;
        reset : in Std_logic;
        rdy : in Std_logic_vector( 1 to no_clients );
```

```
        req : in Std_logic_vector( 0 to no_clients * req_bits – 1 );
        s_done : in Std_logic;
        grant : out Integer range 0 to no_clients );
end Scheduler;
```

This entity corresponds to the scheduler component shown in section 4.5.2. It is parameterized with the number of clients, no_clients, that can be handled by the scheduler and with the width, req_bits, of the request encoding.

The scheduler receives the following inputs:

- The clock signal, clk, and the reset signal, reset.
- The rdy signal, whose *i*-th bit is '1' if the request of the *i*-th client can be accepted, and '0' otherwise.
- The req signal, organised as a one-dimensional array that carries the requests from all the clients. The request of the *i*-th client is encoded in the slice req((i–1)*req_bits **to** i*req_bits–1). A two-dimensional organisation or an array of request vectors would be more natural. However, the first is not synthesizable, and the latter does not allow a parameterization of the request vector width. Hence, a flat representation had to be chosen.
- The done signal from the server, s_done.

The scheduler's output, grant, is the number of the client whose request is granted, or the value zero if there is no acceptable request or no request at all.

A second entity is defined to represent scheduling for guard expressions which only depend on the server's state, but not the service input parameters. As discussed in section 4.5.2, this requires an rdy signal whose *i*-th bit is '1' if the request encoded with the binary value *i* can be accepted, and '0' otherwise. The other inputs and outputs are the same as above:

```
entity Scheduler_State_Only is
    ...
        rdy : in Std_logic_vector( 0 to 2**req_bits – 1 );
    ...
end Scheduler_State_Only;
```

The implementations of scheduling strategies are provided as architectures of the scheduler entity, e.g.:

```
architecture Round_Robin of Scheduler is
begin
    ...
end Round_Robin;
```

These implementations can be in Objective VHDL, behavioural VHDL, or RTL VHDL, but must be synthesizable. Alternatively, a gate level implementation can be provided.

The VHDL attribute SCHEDULING allows to specify the scheduling policy to be used for a concurrent object. This attribute can be imported from the library SCHEDULERS, or can be declared locally by the user:

```
attribute SCHEDULING : String;
```

A scheduling policy is chosen for a concurrent object by specifying a scheduler architecture name as the string value of the object's SCHEDULING attribute; for example:

```
attribute SCHEDULING of buffer_obj : signal is "Round_Robin";
```

5.4.3 Implementation of schedulers

The following scheduling policies, respectively scheduler architectures, have been implemented and are included in the SCHEDULERS library:

- Static_Priority: For each client, a unique priority is defined. Of all acceptable requests, the one that comes from the highest-priority client is granted.
- Round_Robin: The clients are organised in a circular order. The search for the next request to be served begins from the client after the client whose request has been served before. The first acceptable request found is executed.
- Enhanced_Round_Robin: This strategy takes into account clients that have a temporarily unacceptable request. If such client is found during the round robin scheme, it is memorised and given the highest priority next time; i.e., the next search starts from this client. This helps to avoid starvation of requests that are disabled by other requests (cf. section 4.5.3).

The source code of the above schedulers is listed in appendix F. It should be emphasized that the user can implement additional scheduler architectures such as the ones explained below. Likewise, it would be possible to integrate scheduler entities with modified interfaces, e.g. for supplying external priorities, into the synthesis system.

- Equal_Priority: All requests have the same priority, regardless of the issuing client or the service that is requested. If two or more requests are present at the same time, one of them is chosen arbitrarily. This choice is not to be understood as random in the sense that different services or different clients are preferred at different points in time during system operation. Instead, the synthesizer performs the arbitrary choice statically such

that always the same client is preferred at run-time. Only the static choice of this client is arbitrary. A user can implement a random, dynamic strategy by including a pseudo-random number generator in the scheduler.

- First_Come_First_Serve: The oldest acceptable request is chosen for execution. This requires to queue requests in the order of their arrival. Requests that arrive in the same clock cycle are queued in an arbitrary order.

5.4.4 Example

To illustrate the specification of guard expressions and delegation of service execution to a service implementation for variable objects, we now provide the signal implementation of the methods put and get of class LIFO:

```
for signal

    procedure put( val : in Integer ) is
        variable object_copy : LIFO;
    begin
        guard( not is_full );
        object_copy := this;
        object_copy.put( val );
        this <= object_copy;
    end;

    procedure get( val : out Integer ) is
        variable object_copy : LIFO;
    begin
        guard( not is_empty )
        object_copy := this;
        object_copy.get( val );
        this <= object_copy;
    end;

end for;
```

An analogous implementation of put and get can be provided for class FIFO. Now a buffer object can be used from several, concurrent client objects (cf. section 5.3.6) as shown in the following code excerpt. The buffer object is instantiated as a signal and a user-defined scheduling policy is specified. Two producers and one consumer are instantiated. The producers are connected to their server object by port mapping. The consumer, on the other hand, receives the server upon invocation of its consume method. Note that each of

the three clients receives its individual thread of computation from one of the three different processes that invoke the clients' methods.

```
signal buffer_obj : Buffer_t'CLASS;
use SCHEDULERS.DECLARATIONS.SCHEDULING;
attribute SCHEDULING of buffer_obj : signal is "My_Own_Policy";

signal producer_obj_1   : Producer port map( server => buffer_obj );
signal producer_obj_2   : Producer port map( server => buffer_obj );
signal consumer_obj     : Consumer;
...
-- in some process:
consumer_obj.consume( server => buffer_obj );
-- in another process:
producer_obj_1.produce;
-- in yet another process:
producer_obj_2.produce;
```

5.5 Objective VHDL tool architecture

Synthesis of Objective VHDL models has to be integrated into the existing Objective VHDL tool architecture [125]. An overview of a complete synthesis flow is shown in figure 27:

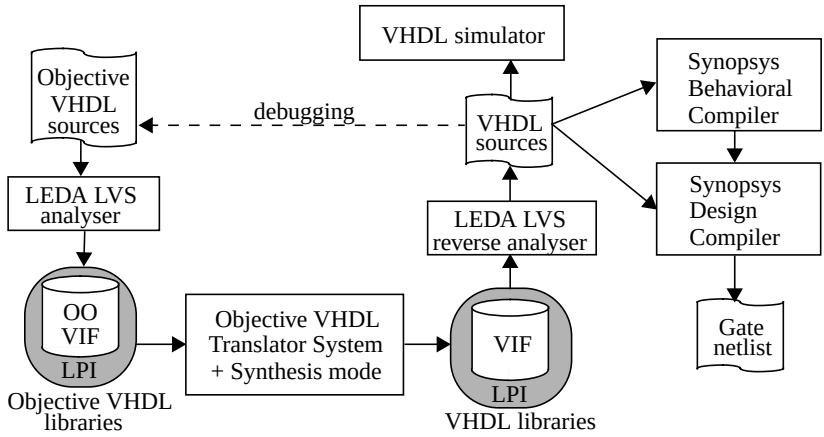


Fig. 27: Objective VHDL tool architecture

The Objective VHDL tool architecture is based on the LEDA VHDL System (LVS) available from LEDA S.A. The LVS is a VHDL compilation and library management system. It has been extended for Objective VHDL.

Objective VHDL source code is first analysed with the LVS analyser [93]. This compiler front-end performs lexical, syntactic, and semantic analysis. If an Objective VHDL model is legal in the sense of the language definition, it is stored in a library according to VHDL's library concept.

The library format, OO-VIF (Object-Oriented VHDL Intermediate Format), is a data structure that represents the abstract syntax tree of the Objective VHDL source code [95]. This tree is annotated with information that has been gained during semantic analysis. The data structure can be held in memory as well as in a persistent file. Its data definition can be extended by tool builders so as to enable the storage of additional, tool-specific information.

Library contents can be accessed through the LPI (LEDA Procedural Interface) [96]. The LPI allows not only to read, but also to augment and modify model information. These capabilities are used in the Objective VHDL Translator System to transform all object-oriented constructs of a model into equivalent VHDL. The result is a pure VHDL library in the VHDL Intermediate Format (VIF) [94].

The translator system consists of a translation engine and a partial Objective VHDL elaborator. In its regular mode of operation, it generates a VHDL model that is as close as possible to the original Objective VHDL. This maintains the original model's semantics and eases the user's understanding of the model's simulation. The implementation of a synthesis mode allows to generate a VHDL model that is suitable for processing with synthesis tools. The details of synthesis mode optimizations and translation are presented in the following chapters.

It is important to note that translation is performed with an analysed, but not yet elaborated Objective VHDL model. This necessitates performing some elaboration tasks in the translation system. However, elaboration in general is a difficult task as it requires the evaluation of functions which can, e.g., be used for calculating generic values. This, in turn, necessitates the interpretation of VHDL's sequential statements and declarations as these can occur in functions. To avoid the enormous complexity of these tasks, it is important to delegate most elaboration to the elaboration engines of the VHDL tools (simulators and synthesizers) which are used as back-ends after translation.

The information stored in the VHDL library can be transformed back into VHDL source code by using the reverse analyser which is a part of the LVS. These sources can be imported into VHDL simulation environments. Simulation allows to find logical errors that let a model's behaviour deviate from the desired functionality. If this is the case, the Objective VHDL code must be corrected and the translation process has to be iterated, as indicated by the debugging arc in figure 27.

VHDL code that has been generated after synthesis mode translation can be processed with synthesis tools. In this work, the Behavioural Compiler and Design Compiler from Synopsys are used. Synthesis translation takes into account the specific coding styles and workarounds required. The choice has been guided primarily by the availability of these synthesis tools to the author and by preferences of project partners.

The tools involved in the translation process from Objective VHDL into VHDL, i.e., analyser, translation system, and reverse analyser, are integrated under a common shell. Thereby, the details of tool interaction via VIF and LPI are hidden from the users.

5.6 Summary

The use of Objective VHDL facilitates the definition of a system of concurrent objects and their communications. We have shown how classes and objects can be defined in Objective VHDL under utilisation of inheritance and polymorphism. The combination of the object-oriented features with VHDL's qualities for static hardware description allows us to model objects and their communication channels in the static way required by the meta-model presented in chapter 4.

Moreover, we have devised an integration of the concept of guarded methods into our approach. Since Objective VHDL does not currently provide guards, an integration via a package has been suggested that enables us to specify guard expressions and extract them for the purpose of synthesis. Thanks to the guarded methods, inter-object communication between concurrent objects is as simple as a subprogram call prefixed by the name of an object, i.e., a signal or port declared with an object-oriented type.

The hardware implementation of all these concepts is to be addressed in the following chapters. We first present the application of data flow analysis techniques for determining optimization potential related to object-oriented constructs and their use.

Chapter 6

Analysis and Optimization of Object Systems

The flexibility of object-oriented modelling, permitting the addition and modification of functionality and the extension of existing systems, is traded in for a certain overhead. Concepts such as polymorphism and dynamic binding require run-time mechanisms for their implementation. In software, these are virtual table lookup operations, for instance. Similar overhead can be observed in a hardware implementation as we will explain in the following section.

To reduce the overhead, optimizations may be performed. This, however, requires to collect information in a preceding analysis step. We will apply data flow analysis techniques in order to narrow down the set of classes (or tags) that can become relevant at certain places during system operation. To this end, we present syntax-directed data flow analysis for extracting information on how tags are propagated in a system. In a further step, we utilise fixed-point iteration techniques in order to compute the worst case distribution of tags. Similar techniques are developed for determining tags which do not need to be distinguished, leading to further optimization potential.

6.1 Optimization of polymorphic objects

In this section, we illustrate the need for optimizations with the example of data types present in a microprocessor model. We outline approaches for determining their propagation in the system and explain the optimizations which are enabled thereby.

6.1.1 Motivation

As defined earlier, a polymorphic object must be able to take on the class membership and state of its root class and any class derived from it which is not abstract. The encoding of tags and states has been defined so that this is possible, and dynamic binding takes into account all redefined versions of a method. However, this is not always needed during system operation since some polymorphic objects might not take on all possible class memberships. For instance, consider a reuse library with a large tree of derived classes which are used in different projects. It is likely that, even if some classes are shared between projects, not all these classes are used in every single project. If a polymorphic object of the base class of this inheritance hierarchy is synthesized to be compatible with all these classes, a large hardware overhead results.

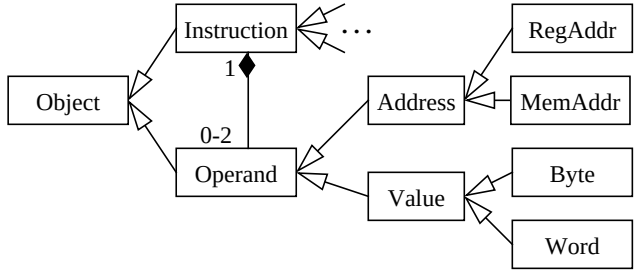


Fig. 28: Inheritance hierarchy of microprocessor data types

Another example are the data types that may occur in a single microprocessor model (see figure 28): Basically, there are instructions and operands. An *Instruction* object may have zero, one, or two operands as exclusively owned sub-objects. An *Operand* may be an *Address* or a *Value*. An *Address* may address a register (*RegAddr*) or a memory location (*MemAddr*). Likewise, there are different types of values, e.g. *Byte* and *Word*.

Polymorphism comes into play when related classes have to be handled in a uniform way. For instance, an attribute, *op*[], that represents the operand sub-objects of an instruction may either be an address or a direct value. Hence, it must be able to take on the state not only of class *Operand*, but of all classes derived from *Operand*, too. Similarly, we want to store both, instructions and operands, in a memory. To this end, we derive both from a common parent class, *Object*, and declare the memory so as to store instances of class *Object* and all derived classes (cf. figure 29). An instruc-

tion register is another example; it should be able to contain any concrete instruction such as *LOAD* or *ADD* derived from class *Instruction*.

Conceptually, a memory declared polymorphic with root class *Object* must be able to store also any class derived from *Object*. However, some classes might, according to the data flow, occur only in some part of the system. For instance, in the memory, and in the register file as well, there never is a register address if this class only occurs as part of an instruction. Or, in a RISC processor [65], the operand within an instruction may only be of class *RegAddr*, not *MemAddr*. The aim of the following considerations is to utilise the resulting optimization potential.

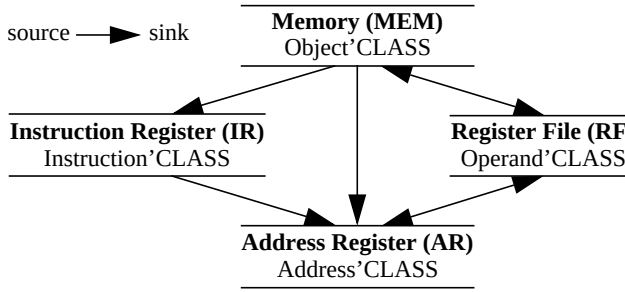


Fig. 29: Data flow in a microprocessor model

6.1.2 Analysis approaches

The set of classes which actually have to be represented by a polymorphic object can be narrowed by a couple of approaches discussed in this section. The approaches are different in sophistication and represent a trade-off between analysis complexity and effectiveness.

The simplest analysis method is to determine all classes that are instantiated in a model. This can be done easily by inspecting the class types used in the declaration of objects. Only these classes, of which objects actually appear in a model, have to be considered when synthesizing a polymorphic object. This is still overly pessimistic since individual polymorphic objects may be limited to different, more narrow class memberships.

The global approach can be refined by considering the distribution boundaries of objects. For instance, an Objective VHDL design entity encapsulates its interior by allowing data transfer only via its ports. Hence, a class type used for declaring an object inside an entity can make an impact outside

that entity only if an interface port is declared with the class type, a class-wide type that includes the class, or a composite type that includes an element of the class. An analysis method could differentiate the sets of class types used in such encapsulated components of a model.

The most detailed approach considers each polymorphic object individually. The distribution of class memberships has to be determined by following the assignments to polymorphic objects. This can be done in the form of static analysis by applying data flow analysis techniques. While these still incorporate worst-case considerations, results much more precise than from the other approaches can be expected. Therefore, data flow analysis of polymorphic objects is pursued in the following sections.

It would be possible to gain class membership information by profiling model execution (simulation), too. However, we can hardly ensure that simulation is exhaustive, i.e., that all possible assignments to polymorphic objects are executed. Hence, this method can be too optimistic and lead to failures when the synthesized system runs into a situation that has not been simulated. This is the reason for not implementing a profiling approach.

6.1.3 Optimization potential

Assume that the set of class memberships which a polymorphic object obj_i with root class R can actually take on during system operation is T_i. This allows to implement the following optimizations, individual to an object, in the synthesized model:

- Instead of all non-abstract classes derived from R, including R (cf. equation 23), the state space of the polymorphic object has to comprise only the classes T_i:

$$S(obj_i) = \dot{\bigcup_{c \in T_i}} \{c\} \times S(c) \subseteq Types \times Values \tag{56}$$

- State space encoding (cf. equation 38–41) needs to consider only the classes in T_i. This helps to save state bits, particularly if unused leaf classes with many attributes are omitted from the encoding. Moreover, less tag bits may be required:

$$b(obj_i) = b_{Tag}(T_i) + max\{b(c) \mid c \in T_i\} \tag{57}$$

- The latest redefinition of a method with respect to a class C may not have to be implemented for obj_i if the object cannot actually become a member of class C, i.e., $C \notin T_i$. Note, however, that the same method version may be the latest redefinition for another class. Hence, a method redefinition

can be omitted from implementation only if it is not the latest redefinition with respect to any class out of T_i.

6.2 Data type analysis

We call the task of determining T_i, the set of classes whose membership an object obj_i may have to take on during system operation, data type analysis. The first step towards data type analysis is to construct a graph that represents the propagation of class types via assignments. Classic data flow analysis techniques are applied to reduce the pessimism in this process.

6.2.1 Data flow analysis techniques

Data flow analysis (DFA) has been developed for application in optimizing compilers of programming languages [3]. One major task of data flow analysis is to find the set of *definitions* of a variable that reach a location where the variable is used. A definition, in DFA terminology, is the potential execution of an assignment to a variable. It is reaches (is effective at) a location in a program (program point) if a program path from the definition to that location exists on which no other assignment to the same variable invalidates (kills) the previous definition. Note that multiple definitions can be effective at a location to which multiple paths exist.

In structured programs, i.e., programs that are composed of elementary statements, sequences of statements, conditional statements, and loops, the propagation of definitions can be computed by induction over the program structure. Formulations for more general programs with arbitrary jumps are based on a control flow graph representation. Propagation of definitions is performed while reducing the control flow graph by applying certain transformations (interval analysis). Splitting of vertices or the application of fixed-point iteration techniques allows to deal with otherwise irreducible control flow graphs at the cost of increased analysis effort. An overview of these techniques can be found in [81].

The knowledge of reaching definitions enables various optimizations, including the extraction of invariants from loops, removal of redundant copies, and elimination of interdependent induction variables. DFA has been applied in order to gain structural type information from programs in untyped languages, too [3]. This problem has been identified as difficult as the set of possible data structures is infinite. For instance, an untyped variable may be

assigned a scalar value, an array of scalars, an array of arrays of scalars, and so on infinitely.

The problem covered in this work is simpler, but different: The declaration of polymorphic variables with a class-wide type already narrows its possible types to the root class of polymorphism and the finite set of classes derived from it. Furthermore, type identifiers, but not the type structure, need to be considered. Still, our specific DFA application has not been addressed before to the best of our knowledge. It is irrelevant for compilers of object-oriented programming languages thanks to their reference semantics of object polymorphism. Pointers are of the same size regardless of what they reference. Only the hardware-specific value semantics makes our analyses necessary to reduce the size of a polymorphic object's state vector.

We apply standard DFA techniques to determine definitions of polymorphic objects that are used only in a single sequential model segment. In Objective VHDL, these objects are declared as variables. Their treatment is tailored to our need, which is to build up a definition-use graph that represents the propagation of class types which is caused by assignments. DFA helps to differentiate between individual definitions and, hence, different class membership sets of the same polymorphic object when this object is used at different locations.

The analogue is not possible for concurrently used objects, which are declared as signals in Objective VHDL. The interleaving of assignments from several, concurrent processes cannot be predicted statically. Hence, it is hardly possible to narrow down the set of definitions which reach a certain location in a model's source code. We must make the worst-case assumption that all definitions of a concurrently used object are effective at any location. Pessimism could be reduced by applying DFA to the definitions and uses of concurrent objects in the same process. However, definitions from other processes must still be considered as effective at any location of the process under examination. Respective analysis methods have been developed in [69][70] for signals which are assigned only from a single process. They take into account the complication that signal assignments with delta delay do not become effective immediately after the assignment, but after the next wait statement in the control flow. However, assignments with larger-than-delta delay are not covered.

6.2.2 Notations

Basic to the presentation in this work are the following terms:

- The set of all polymorphic objects that are used sequentially only, e.g., Objective VHDL variables of a class-wide type:

$$\Omega^{s}_{poly} = \left\{ obj^{s}_{i} \,\middle|\, i = 1, \ldots, n_s \right\} \subseteq \Omega . \tag{58}$$

- The set of polymorphic objects that are potentially used from different concurrent domains, e.g., Objective VHDL signals of a class-wide type:

$$\Omega^{c}_{poly} = \left\{ obj^{c}_{i} \,\middle|\, i = 1, \ldots, n_c \right\} \subseteq \Omega . \tag{59}$$

- The set of all definitions of polymorphic objects, *Defs*, that occur in the model under analysis.
- The *definition vector*, $D \in \wp(Defs \cup Types)^{n_s}$, whose *i*-th element D_i is the set of definitions or types of the *i*-th sequential polymorphic object, $obj^{s}_{i} \in \Omega^{s}_{poly}$, that reach the program point under examination. The inclusion of types allows to propagate a type instead of a definition if the type can already be determined during DFA.
- The empty definition vector, $D = \varnothing$, whose elements are empty sets.
- The *constraint vector*, $C^{s} \in \wp(Types)^{n_s}$, whose *i*-th element is the maximal set of types to be contained by the sequential polymorphic object obj^{s}_{i}, i.e. its root class of polymorphism and all non-abstract derived classes.
- An analogous constraint vector, $C^{s} \in \wp(Types)^{n_c}$, defined for the concurrent polymorphic objects.

In the remainder of this chapter, only class types and derived class types need to be considered so that *Types* can be narrowed accordingly.

During DFA, an edge- and vertex-valued directed graph $G = (V, E, T, \mu)$ with the following properties is constructed:

- The vertices, *V*, are concurrent objects and definitions of sequential objects, plus a dedicated entry vertex *e*: $V \subseteq \Omega^{c} \cup Defs \cup \{e\}$.
- The edges, $E \in V \times V$, represent assignments by which types are propagated.
- Edges are attributed with type constraints. These are defined by the mapping $\mu : E \rightarrow \wp(Types)$ from the set of edges into the power set of types. The constraint $\mu(v_1, v_2)$ is the set of types that can be propagated from vertex v_1 to v_2 via the edge (v_1, v_2).

- Vertices are attributed with type sets by a mapping $T : V \rightarrow \wp(Types)$. The value $T(v)$ is the set of possible class memberships of *v*. Its computation is addressed in section 6.3.

For example, the following sets and vectors may correspond to figure 28 and figure 29:

- Ω^{s}_{poly} = {AR}, Ω^{c}_{poly} = {IR, MEM, RF}.
- *Types* = {Object, Instruction, Operand, Address, RegAddr, MemAddr, Value, Byte, Word }.
- C_{RF} = {Operand, Address, RegAddr, MemAddr, Value, Byte, Word} (C_{RF}: element of *C* corresponding to RF).
- *T*(RF) = { MemAddr, Byte, Word } as analysis result (see section 6.3.5).

6.2.3 Definitions

This section describes the construction of the graph *G* according to the definitions found in a model. A definition *d*, in the sense of DFA, of a polymorphic object can be caused by the assignment of another object's state, the assignment of the evaluation result of a class-typed or class-wide expression, and the association of the object as an actual parameter to a formal output parameter of a subprogram or method. Respective definitions of a polymorphic object, obj_i, are depicted in figure 30. We first investigate the direct assignment from object to object.

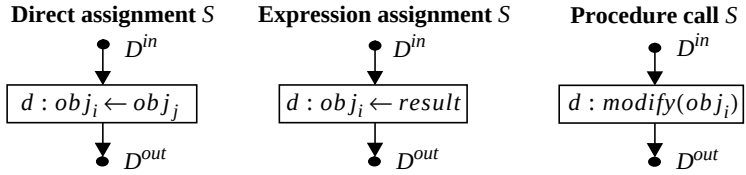


Fig. 30: Definition of an object

As mentioned previously, individual definitions are distinguished only for sequential objects. Hence, sequential and concurrent target objects and source objects are treated in different ways. We first consider a sequential target object, $obj_i \in \Omega^{s}_{poly}$, and distinguish the following cases according to the source of assignment:

1) The source is a non-polymorphic object of class *t* (see figure 31, case 1). The assignment defines the target to be a member of class *t*. Any previous definitions are killed. Hence, we update the corresponding element, D_i, of the definition vector with the class type *t*. A definition vertex for repre-

senting the assignment of class types still to be determined does not have to be created.

2) The source is a concurrent polymorphic object, $obj_j \in \Omega^c_{poly}$ (see figure 31, case 2). Individual definitions of this object are not distinguished; there exists only a global vertex, $obj_j \in V$, that comprises all its definitions. An edge from this vertex to the current definition, *d*, is established and attributed with the target object's constraint:

(60) $(obj_j, d) \in E$ with $\mu(obj_j, d) = C_i^s$.

As the current definition, *d*, becomes effective for the target object, the corresponding element of the definition vector is overwritten: $D_i := \{d\}$.

3) The source is another sequential polymorphic object, $obj_j \in \Omega^s_{poly}$ (see figure 31, case 3). Its definitions are stored in the element D_j of the definition vector. DFA actions depend on these definitions:

- If no definitions of the source object are known yet ($D_j = \emptyset$), e.g. at the start of analysis or during the analysis of loops, a vertex that represents the definition, *d*, of the target object is created. The target object's definition set, D_i, is updated with this definition: $D_i := \{d\}$.
- If D_j includes only class types, i.e., the class types of all definitions of the source object are known, no definition vertex needs to be created. Only the class types, D_j, with which the source may be defined are adopted as the definitions, D_i, of the target object to the extent its constraint, C_i^s, allows an assignment of these types: $D_i := D_j \cap C_i^s$.
- If D_j includes definitions, a new vertex that represents the definition under analysis, $d \in V$, is created unless it already exists in the graph. Edges are created from each source definition, $s \in D_j$, to *d*, and are attributed with the target object's constraint, C_i^s:

(61) $\forall s \in D_j \cap Defs : (s, d) \in E$ and $\mu(s, d) = C_i^s$.

Note that the intersection with *Defs* leaves all types that might be in D_j out of consideration. Instead, these types are added to the set of known types of definition *d*. This requires to create an edge (*e*, *d*) from the entry vertex *e* to vertex *d*, attributed with the initial value $\mu(e, d) := \emptyset$, unless the edge already exists in the graph. The types, determined by intersecting D_j with C_i^s, are added to this edge: $\mu(e, obj_i) := \mu(e, obj_i) \cup (D_j \cap C_i^s)$. Finally, the definition *d* becomes the new effective definition of the target object by overwriting the corresponding element of the definition vector: $D_i := \{d\}$.

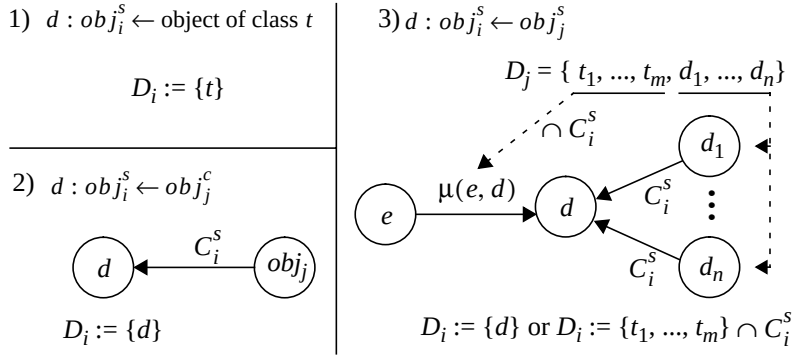


Fig. 31: DFA actions for sequential polymorphic object as assignment target

The definition of a concurrent polymorphic object, $obj_i \in \Omega^c_{poly}$, is handled similarly with respect to the different possible sources. However, as no individual definitions are considered, a global vertex representing all definitions of obj_i takes the place of an individual definition vertex. Moreover, the updating of a definition vector is not applicable. The definition variants (see figure 32) can be summarised as follows:

1) If the source is a non-polymorphic object of class *t*, this class type is added to the target object's known types: $\mu(e, obj_i) := \mu(e, obj_i) \cup \{t\}$.
2) If the source is a concurrent polymorphic object, an edge from the source to the target object is created, being attributed with the target's constraint.
3) If the source is a sequential polymorphic object, all types with which it may be defined and which comply with the target object's constraint are added to the target's known types: $\mu(e, obj_i) := \mu(e, obj_i) \cup (D_j \cap C_i)$. All definitions of the source objects are linked to the target object via an edge that is attributed with the target's constraint.

We now take into account definitions that involve the assignment of an expression or output parameter. An expression can be handled like a function call, particularly in Objective VHDL, where all operators that occur in an expression are defined as functions. If the function's return type or output parameter type is non-polymorphic, the definition of the target object is dealt with as above. For a function or parameter that returns a class-wide type we could assume the worst case, that is, the target may be defined with any class that belongs to the class-wide type. A less pessimistic estimation requires the examination of the expression or procedure to narrow down the set of possible return types.

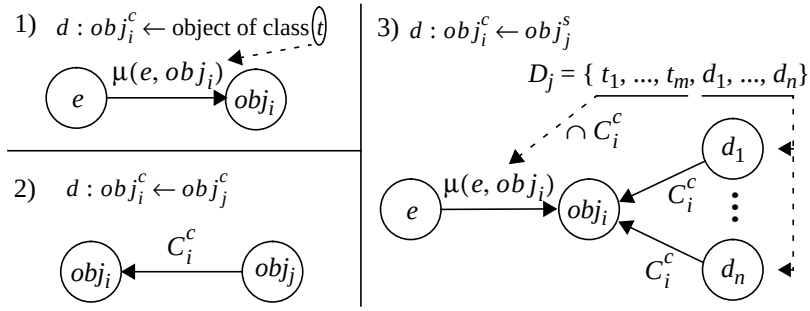


Fig. 32: DFA actions for concurrent polymorphic object as assignment target

This leads to interprocedural data flow analysis, which means to observe the subprogram call hierarchy, to provide inputs for the data flow analysis of called subprograms, to analyse these called subprograms, and to transfer DFA results back for further consideration at the location where the subprogram has been invoked. Respective techniques are explained for Objective VHDL's hierarchy structures in section 6.2.6.

6.2.4 Consideration of control flow

In the previous section, only single definitions have been taken into account. We now consider the control flow of the program for global data flow analysis. To this end, we understand the effect of a statement as a transformation of the definition vector. Let D^{in} be the definition vector at the location before a statement. The analysis of a statement S yields the definition vector D^{out} which is valid immediately after the statement: $D^{out} = S(D^{in})$.

The analysis of a definition of a sequential polymorphic object has been examined in the previous section. The view is now extended to control structures of which structured sequential algorithms are composed: sequences of statements, conditional statements (alternatives), and loops (see figure 33). Recall that a similar analysis cannot be made for concurrent objects, which are handled in a more pessimistic way by not distinguishing definitions.

- In a sequence, *S*, of a statement *S1* followed by *S2*, the definition vector created by the first statement, $D = S1(D^{in})$, is the input to analysis of the second statement. Hence, the result of the sequential analysis of both statements is obtained as $D^{out} = S(D^{in}) = S2(D) = S2(S1(D^{in}))$. An extension to sequences of *n* statements, $S1, \ldots, Sn$, is obvious:

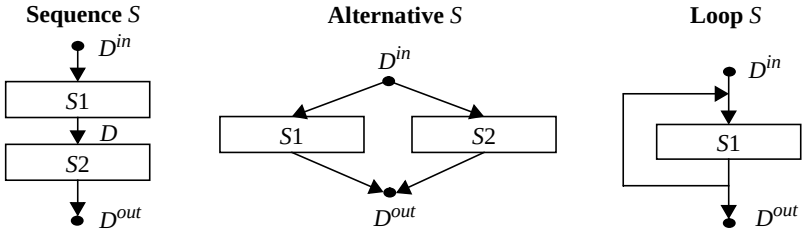


Fig. 33: Control flow in structured programs

(62) $D^{out} = S(D^{in}) = Sn(\ldots S2(S1(D^{in}))\ldots)$.

- The alternative of two statements, $S1$ and $S2$, is described by a control flow that branches to either of the statements, and reconverges thereafter. This corresponds to an if-then-else statement. While only one of the alternatives is taken each time the statement is executed, we cannot statically determine which. Furthermore, both branches may be chosen when considering multiple executions of the alternative statement. Hence, a conservative analysis of the definitions that reach the location after the statement must include the definitions from $S1$ and from $S2$: $D^{out} = S(D^{in}) = S1(D^{in}) \cup S2(D^{in})$, where the union operator is element-wise for the definition vector. DFA can be implemented by analysing $S1$ and $S2$ independently and merging the results. An if statement without else branch can be represented by defining $S2$ as an empty statement with $S2(D^{in}) = D^{in}$. An extension to case statements with n alternatives, $S1, \ldots, Sn$, is possible:

(63) $D^{out} = S(D^{in}) = \bigcup_{i=1}^{n} Si(D^{in})$.

- Finally, a loop statement S shall be considered. The definition vector $S1(\varnothing)$ generated by the statements inside the loop may be propagated back to its start when the loop is iterated. $S1(\varnothing)$ is therefore used in addition to D^{in} as input to the analysis of the inner statement $S1$. The result is the vector of all definitions which may be valid after one or more iterations of the loop. An implementation first analyses the loop interior with an empty input definition vector, merges the generated definitions with the definition vector that is valid before the loop, and uses the result for analysing the iterated statement a second time. A while or for loop, which

may not be entered at all, can be handled by adding an unmodified D^{in} to cover the case of bypassing the loop. D^{out} follows as:

$$(64) \quad D^{out} = S(D^{in}) = S1(D^{in} \cup S1(\varnothing)) \quad \left[\cup D^{in}, \text{unentered loop} \right]$$

We finally remark that **process** statements, which are restarted after their last statement, are handled like infinite loops by Objective VHDL DFA.

6.2.5 Unstructured control flow

VHDL and Objective VHDL provide a couple of statements that do not comply with the notion of structured programs. The **return** statement allows to complete the execution of a subprogram, including a method, by leaving its control flow at an arbitrary location rather than its end. Loops can be quit from within their interior by the **exit** statement. The **next** statement jumps back to the beginning of a loop. Things are further complicated by the possibility to use these statements for jumping across loop boundaries of nested loops as it is shown in the following listing. DFA techniques have to be extended so as to cope with these features.

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

As opposed to arbitrary goto statements, which are not available in VHDL, the loop and subprogram control statements allow to maintain a syntax-directed approach to DFA by adapting a programming trick reported in [3]. Alternatives are to perform fixed point iteration of data flow equations during analysis or to apply interval analysis on an explicitly generated control flow graph. However, as the intermediate format used in this work is an abstract syntax tree, the syntax-directed approach is the most convenient to follow.

Figure 34 shows the control flow caused by a **return**, **exit**, or **next** statement, respectively. All these control statements have in common that they break the normal control flow by jumping to a defined location: the end of a subprogram, end of a loop, or beginning of a loop, respectively. As indicated by the no entry symbol, any following statements cannot be reached via a

path that includes such control statement. They can, however, be reached by alternative control flows suggested by the dotted lines.

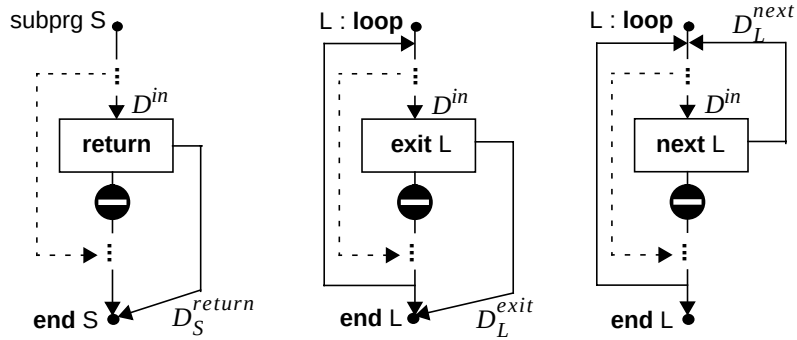


Fig. 34: Control flow of **return**, **exit**, and **next** statements

As program execution is not continued after a control statement, no definitions can be propagated to the following statements. Hence, we specify $D^{out} = \varnothing$ for **return**, **exit**, and **next** statements.

However, the definitions that reach one of these statements are propagated to a known target location. As the control flow to this location does not correspond to the program structure, we cannot express propagation in the framework of syntax-directed analysis. Instead, we store these definitions in global variables for later consideration. This allows us to deal with multiple control statements, too, e.g. multiple return statements of a subprogram:

- For each subprogram *S*, a vector of definitions propagated by **return** statements, D_S^{return}, is defined and initialised empty at the beginning of analysis. All definitions that reach a **return** statement are added to this vector: $D_S^{return} := D_S^{return} \cup D^{in}$.
- In analogy, vectors of definitions propagated by **exit** and **next** statements, D_L^{exit} and D_L^{next} are defined and initialised empty for each loop *L*. Definitions that reach an **exit** statement are added to D_L^{exit}, and definitions that reach a **next** statement are added to D_L^{next}.
- A **next** statement in a loop with a condition that is evaluated at the beginning of the loop (i.e., VHDL **for** and **while** loops) may in addition have the same effect as an **exit** statement. After the jump to the loop start, the condition may be evaluated as false, and the loop be quit in consequence. Hence, definitions that reach a **next** statement of such a loop must be added to D_L^{exit}, too.

These special definition vectors have to be taken into account when computing the output definition vector of a subprogram or loop, respectively:

- The definitions D_S^{return} are propagated to the end of a subprogram S in addition to the definitions $S(D^{in})$ which reach the subprogram's end via the normal control flow. Hence,

(65) $D^{out} = S(D^{in}) \cup D_S^{return}$.

- Similarly, the definitions D_L^{exit} reach the end of the loop L in addition to those of equation 64. Finally, the definitions D_L^{next} are propagated to the beginning of the loop L, where they add to the definitions that are input to the loop. The DFA equation for loops (cf. equation 64) is therefore rewritten as:

(66) $D^{out} = S1(D^{in} \cup S1(\varnothing) \cup D_L^{next}) \cup D_L^{exit} \quad [\ \cup D^{in}]$

Only unconditional control statements have been considered up to now. VHDL's **next** and **exit** statements, however, can include an optional condition under which they are executed. This can be analysed after a conceptual transformation into an unconditional **exit** or **next** that is nested into an **if** statement:

exit L **when** condition;	=>	**if** condition **then exit** L;
next L **when** condition;	=>	**if** condition **then next** L;

6.2.6 Interprocedural and hierarchical analysis

VHDL offers two sorts of hierarchical modelling to the designer. First, design hierarchy can be described through instantiation of design entities (entity-architecture pairs). These instances are concurrent with respect to another. Second, functional decomposition into subprograms that call other subprograms is provided. We can understand the execution of an invoked subprogram as an instance of that subprogram. As opposed to entity instantiation, a subprogram instance is sequential with respect to the subprogram from which it has been invoked.

With each call to a subprogram or instantiation of a design entity, different actuals may be passed to the formal parameters or ports, respectively. We take this fact into account through an *instance tree* (see figure 35) that corresponds to instantiation and call hierarchies. Each node (instance) has its own data structures for DFA so that different instances can be analysed independently. Global signals declared in packages are taken into account through a list of used packages at the root of the instance tree.

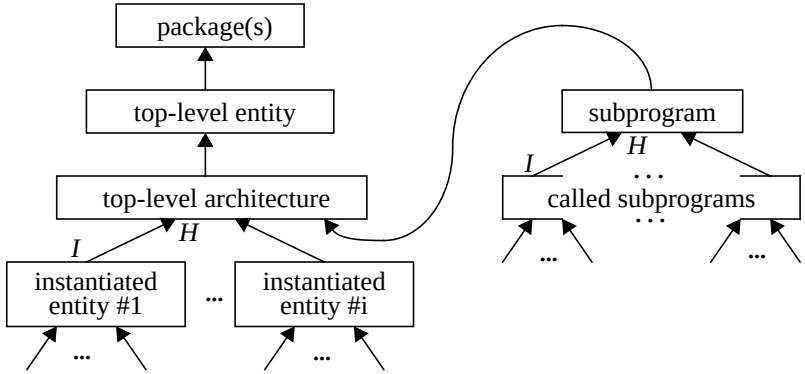


Fig. 35: Handling of structure and call hierarchy through an instance tree

The connection between an instance *I* and the higher level in hierarchy, *H*, is through interface objects (i.e., formal subprogram parameters and entity ports). If such interface object is an input parameter or port, it is defined with the assigned actual parameter. If it is an output, its association with an actual parameter causes a definition of that actual. Bidirectional interface objects, i.e., ports or parameters declared as **inout** in Objective VHDL, is dealt with by two definitions, one in each direction. Thereby, connection between the instances is established. The flow of data types through the interface objects is computed by a fixed-point iteration algorithm (see section 6.3).

Recursive subprogram calls need special consideration. Recursion is synthesizable if its depth can be determined during static elaboration. In this case, the synthesis tool evaluates all recursive calls, allocates the required resources, and synthesizes a circuit that works without actually implementing a recursion mechanism. The application of such an elaboration strategy would allow us to build up a call tree of the instances (invocations) of a synthesizable recursive subprogram and to analyse them individually. However, an elaborator is not available in the Objective VHDL tool environment.

An analysis of recursive subprograms is still possible by allowing cycles in the instance graph. Thereby, all recursive invocations of a subprogram are analysed together and the results effectively merged. A subprogram that calls itself, directly or indirectly, would contribute to the definitions of its own input parameters and would receive definitions via its output parameters. While this simplifies analysis, results become more pessimistic as for each individual instance the definitions of all other instances are considered, too.

6.2.7 Example

We now demonstrate sequential DFA with a simple example that emphasizes the benefit of considering individual definitions of sequential objects. The following pseudo code excerpt belongs to the simplified microprocessor model presented in section 6.1.1. It implements register-indirect addressing:

```
d1: AR  ←  IR.getOp(1);   – – D_AR = { d1 }
d2: AR  ←  RF[ AR ];      – – D_AR = { d2 }
d3: AddrBus  ←  AR;
```

In the first line, the first operand of the instruction register, IR, is loaded into the address register, AR. We know that this operand is a register address as register-indirect addressing is being implemented. The value stored in the addressed register of the register file, RF, is loaded into AR (second line). In the register-indirect addressing scheme, we know that this value is a memory address. Finally, in the third line, this value is assigned to the address bus of the main memory. Note that the mentioned information about the values' types is not explicitly defined in the model. It is known to the designer, and it can be extracted by type propagation as we show in section 6.3.5.

We assume that AR is a sequential object; the other objects be concurrent. Sequential DFA determines that AR is defined by definition d1 after the first line, and by definition d2 after the second line. During this analysis, the data type flow graph shown in figure 36 is generated. The first definition, d1, is represented by a vertex that is reached by an edge from an interface object that represents the return value of the function method getOp. Analysis of this function is performed using our interprocedural DFA concepts. The second definition, d2, results in a vertex with an ingoing edge from the source object, RF. Both edges mentioned are attributed with the type constraint of their target object, AR, which is declared as a polymorphic object with root class Address.

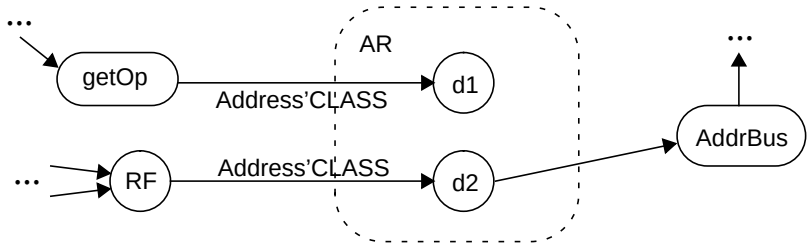


Fig. 36: Data type flow graph generated by DFA

The third definition, d3, has a concurrent polymorphic target object. Hence, no individual vertex is created for this definition. Instead, an edge goes from the effective definition of the source object, d2, to a vertex that represents the target object, AddrBus. Thanks to DFA, we know that only values from the register file are propagated to the address bus, but no instruction operands. Without DFA, AR would be handled as a single vertex. Due to this more pessimistic assumption, instruction operands would reach the address bus when type propagation is computed, even if this does not happen during system operation.

6.3 Determining type information

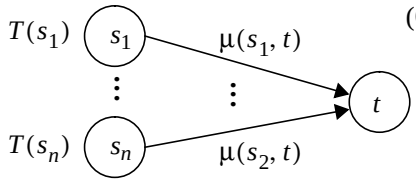
While syntax-directed DFA has enabled us to find the propagation paths (definition-use chains) of class types without a fixed-point iteration, the problem of determining the flow of type information over these paths requires fixed-point techniques. We present the mathematical framework behind fixed-point iteration and show that it is applicable to our problem. In consequence, a standard algorithm can be utilised and has been implemented.

6.3.1 Propagation of types

The graph resulting from data flow analysis allows to follow the propagation of types due to the use of definitions of polymorphic objects. To represent this flow of type information in the graph, each vertex d is valued with the set of types, $T(d) \subseteq Types$, which an object may take on due to its definition d. This set is initially empty for any but the entry vertex. Its computation is the matter of this section.

If a definition s of a source object may be used in the definition t of a target object, types that can be taken on by s have to be taken on by t, too. In the graph, this use of s is represented by an edge (s, t). Hence, the type information must be propagated over this edge. Type propagation is, however, constrained by the edge's attribute $\mu(s, t)$, in which the maximum set of types that can flow via the edge is specified. Only the types which are in $T(s)$ and allowed by $\mu(s, t)$, i.e. $T(s) \cap \mu(s, t)$, reach the vertex t.

Multiple edges may lead to a vertex, corresponding to several definitions which are possibly used. This information must be combined by a so-called confluence operator to determine the set $T(t)$. For the purpose of type propagation, we choose the union of type sets so as to consider any type of any possibly used definition. This results in equation 67 (see figure 37):



$$(67) \quad T(t) := \bigcup_{(s,t) \in E} T(s) \cap \mu(s,t)$$

Fig. 37: Propagation of data types over flow graph edges

At this point it is important to study the role of entry vertex. This is the only vertex whose type set is not subject of computation, but defined according to the application. We choose the set of all types: $T(e) := Types$. An edge from the entry vertex to a definition vertex is attributed with the set of known types of this definition, $\mu(e,d)$. By propagation of *Types* over this edge, exactly the known types reach the target: $Types \cap \mu(e,d) = \mu(e,d)$. Thereby, we introduce information on known types in a way compliant to a mathematical structure that will be discussed in the next section.

As opposed to definitions, the propagation of types is transitive, i.e., a type may reach a polymorphic object via intermediate stations. This is considered by propagating types that have reached a vertex via an ingoing edge over the outgoing edges to further vertices. As the definition-use graph may have cycles, this becomes non-trivial. We must carefully investigate whether the vertex's type sets can reach a stable state (fixed point) which is not changed by further propagation, whether this fixed point is a meaningful result (e.g., the best possible with respect to optimization), and how this fixed point can be computed. All these questions can be answered with the help of DFA theory.

6.3.2 Type propagation in a data flow analysis framework

DFA theory allows to make statements about a set of problems that can be described as an instance of a so-called data flow analysis framework. This is a mathematical structure originally developed by Kildall [82]. It has been presented, e.g., in [3]. In this work, the more axiomatic presentation in [83] is utilised. It can be summarised and adapted to our terminology as follows:

A *monotone data flow analysis framework* is a pair (L, F), where L is a semilattice with partial order $\geq$, supremum operator $\vee$ and top element $\top$, and F is a set of transfer functions $f : L \to L$ with the following properties:

- Monotony: $\forall f \in F \ \forall x, y \in L : x \geq y \Rightarrow f(x) \geq f(y)$.

- Identity function: $\exists f \in F\ \forall x \in L : f(x) = x$.
- Closedness under composition: $\forall f_1, f_2 \in F : f_1 \circ f_2 \in F$.
- $\forall l \in L\ \exists f \in F : f(\top) = l$.

In a *distributive* framework, $\forall f \in F\ \forall x, y \in L : f(x \vee y) = f(x) \vee f(y)$ must hold in addition. Distributivity implies monotony.

To be a *partial order*, the relation $\geq$ must be

- reflexive: $\forall x \in L : x \geq x$,
- antisymmetric: $\forall x, y \in L : x \geq y \wedge y \geq x \Rightarrow x = y$, and
- transitive: $\forall x, y, z \in L : x \geq y \wedge y \geq z \Rightarrow x \geq z$.

For the *supremum* $\vee$ to exist, there must be

- an upper bound, u, for any two values: $\forall x, y \in L\ \exists u \in L : u \geq x \wedge u \geq y$,
- which is a least upper bound: $\forall x, y, u' \in L : u' \geq x \wedge u' \geq y \Rightarrow u' \geq u$, i.e., any other upper bound u' is greater than or the same as u.

If for any two values, x and y, a smallest upper bound $u(x, y)$ exists, we can define the supremum operator $\vee : x \vee y \equiv u(x, y)$. This makes the partially ordered set L a *semilattice* [61]. If in addition the greatest lower bound (infimum) exists, L becomes a *lattice*. Any lattice is a semilattice, too.

A *top element* is an element $\top \in L$ with $\forall x \in L : \top \geq x$. A *bottom element* is an element $\bot \in L$ with $\forall x \in L : x \geq \bot$. These elements need not exist in a lattice. However, any lattice can be embedded in another lattice with artificial top and bottom elements if required.

DFA theory requires a confluence operator, for which we use the supremum. A dual presentation would be possible based on the infimum. A transfer function describes the operation invoked when information is propagated over an edge of the analysed graph. The set L includes the data values that are propagated. The top element corresponds to a DFA result that makes optimization impossible, while the bottom element represents the case with most optimization potential. Hence, for the purpose of type propagation, we define:

- $L \equiv \wp(Types)$, the power set of *Types*, as the values propagated are subsets of *Types*.
- $\geq \equiv \supseteq$
- $\vee \equiv \cup$ as set union is our confluence operator.
- $F \equiv \{ f_T : \wp(Types) \to \wp(Types) \mid T \in \wp(Types) \}$ where for each transfer function holds $f_T(X) := X \cap T$ for all $X \in \wp(Types)$. This allows to represent the intersection with a constraint set $\mu(s, t)$ (cf. equation 67) as $f_{\mu(s,t)}$.
- $\top \equiv Types$ because a polymorphic object that must be able to take on any class membership is the worst case with respect to optimization.

- $\bot \equiv \varnothing$ since a polymorphic object that takes on no class membership is the best case with respect to optimization—it can be discarded.

It is well-established that any power set $\wp(S)$, ordered by set inclusion, is a lattice (hence, a semilattice, too) with supremum defined by the union of sets, top element *S*, and bottom element $\varnothing$ [61]. Furthermore, we know that intersection, i.e. the transfer function, is distributive over union. The remaining properties are easy to show: f_{Types} is an identity function as $f_{Types}(X) = X \cap Types = X$. F is closed under composition as $f_B(f_A(X)) = (X \cap A) \cap B = X \cap (A \cap B) = f_{A \cap B}(X)$. Finally, it holds $f_A(\top) = A \cap Types = A$. Hence, our type propagation problem fulfils the axioms of a distributive data flow analysis framework. This allows to apply the results of this theory.

6.3.3 Fixed-point iteration

Given that the information propagated through the flow graph *G* fits into the model of a distributive data flow analysis framework, a unique least fixed point exists as the solution of the data flow equations. This fixed point can be computed by the following algorithm, which has been adapted from [83] to our terminology. Its pseudo code notation is:

```
(1)   procedure FPI( G = (V, E, T, μ) ) is
(2)       T := ( entry => Top, others => Bottom );
(3)       change := true;
(4)       while change loop
(5)           change := false;
(6)           for each t in V – {entry} loop
(7)               temp := Bottom;
(8)               for each ( s, t ) in E loop
(9)                   temp := temp ∨ f_μ(s,t)( T(s) );
(10)              end loop;
(11)              if not temp = T(t) then
(12)                  T(t) := temp;
(13)                  change := true;
(14)              end if;
(15)          end loop;
(16)      end loop;
(17)  end;
```

The algorithm initialises the type set of the entry vertex with the top element (line 2) so as to allow the introduction of known types as explained in section 6.3.1. The type sets of all other vertices are initialised with the bot-

tom element in order to approach the least fixed point from below. A termination flag, change, is initialised in line 3.

The body of procedure FPI is iterated as long as the type sets change (line 4). For each vertex t (line 6) which is not the entry vertex, the type set is computed in lines 7–10 as the supremum over the values of the transfer functions of all ingoing edges (s, t). The result is stored in temp. Only if it is different from the result of the previous iteration, the value of temp is taken as the new type set T(t) and the change flag set to true (lines 11–14).

There are several ways to speed up this algorithm [6]:

- Static single-assignment form helps to reduce the total number of edges by collecting multiple definitions of an object that reach the end of a conditional statement in a single definition of an artificial intermediate variable so that subsequent uses of the object cause only one ingoing edge.
- We can maintain a work list of vertices whose predecessor's type set has changed. Only these vertices have to be considered in each iteration of the loop at line 7.
- Vertices can be visited according to the main flow of information. Then, if the definition-use graph has no cycles, a single pass of the outermost loop, visiting vertices in topologically sorted order, suffices. In the presence of cycles, we can still attempt to visit as many vertices as possible before their successors.

Independent of these improvements, the worst case complexity of the fixed-point iteration problem is $O(n^3)$ under the assumption that supremum and transfer function can be computed in constant time. This is possible when the problem can be mapped onto a bit vector representation and bit-wise machine operations are exploited.

Yet, sources [83] report that in practice the run-time of fixed point iteration implementations is approximately linear in program size. We have made the same observation with an implementation for Objective VHDL.

6.3.4 A concrete implementation

Our implementation [128, 150] of syntax-directed computation of definition-use chains and the following fixed-point iteration for type propagation implements the basic fixed-point iteration algorithm as presented in the previous chapter. A bit-vector representation has not been pursued; instead, library routines for a higher-level representation of sets are employed. Neither a work list approach nor static single-assignment form have been implemented. This is because, rather than focusing on an optimization of the analysis algo-

rithm, emphasis has been put on a wide support of the Objective VHDL language features (cf. section 6.2.5).

To measure our implementation's run-time performance development with increasing model size, a scalable synthetic benchmark has been developed. It is based on a design entity with 10 internal polymorphic objects, 5 of them sequential and 5 concurrent, and 2 bidirectional polymorphic port signals (interface objects). The entity contains 32 assignments between these objects, organised in a way that cycles of maximal length, 6, occur among the sequential as well as the concurrent objects separately. Furthermore, assignments between the sequential, concurrent, and interface objects are included. This entity has been repeatedly instantiated and interconnected as a tree structure so that information is passed among the instances in both directions via their interface objects.

Run-time results obtained from execution on a Sun Sparc 20 workstation are presented in figure 38 for 10 instances of the problem, $a1$ to $a10$, with an entity instantiation depth of 1 to 10. As we see, fixed point iteration (FPI) takes significantly more time for large problems than syntax-directed DFA. However, even for the largest problem, the total run-time of 12 minutes is still in the range acceptable in hardware synthesis. This holds particularly if we consider that 10,230 objects of, say, 16 bits each, would correspond to over 160,000 flip-flops for state storage.

	instances	polym. objects	assign-ments	interface objects	DFA [s]	FPI [s]
$a1$	1	10	32	0	0.05	0.02
$a2$	3	30	96	4	0.07	0.12
$a3$	7	70	224	12	0.12	0.57
$a4$	15	150	480	28	0.18	2.00
$a5$	31	310	992	60	0.35	6.02
$a6$	63	630	2,016	124	0.70	17.05
$a7$	127	1,270	4,064	252	1.37	45.48
$a8$	255	2,550	8,160	508	2.80	118.02
$a9$	511	5,110	16,352	1,020	6.10	299.55
$a10$	1,023	10,230	32,736	2,044	13.15	731.63

Fig. 38: Optimization run times

Approximately, the number of objects doubles and total run time triples with every row of the table. Hence, run-time increase can be considered linear in the number of polymorphic objects at least for our experimental setup.An investigation reveals that the following points support the reasonable run time behaviour:

- The definition-use chains are pre-computed during the fast syntax-directed analysis phase. All definitions, including those in program loops, are thereby determined before fixed-point iteration. This reduces the number of cycles in the type propagation graph.
- A cycle still occurs when a variable is used before its definition. The more frequent case is that the definition is located before the use in the program text. Syntax directed analysis creates vertices (corresponding to definitions) in top-down program order. The FPI implementation processes vertices in the order they are created, i.e., according to the primary direction of information flow.
- The design hierarchy defined by the instantiation tree of an Objective VHDL model limits the exchange of types in a system. Types can flow into or out of an instance only via its interface objects. This limits the connectivity of the type flow graph, i.e., the number of ingoing edges that must be processed in the innermost loop of the FPI algorithm from section 6.3.3, and possibly, thanks to faster convergence, the number of iterations of the outermost loop, too.

Finally, if a model should turn out too large or involved for our analysis algorithms, it can be split into smaller parts, e.g. the entities that represent its major components. At their boundaries, worst case conditions may be assumed. Alternatively, the user can constrain external type sets as described in the following section. This, however, can lead to inconsistencies if the user's assumptions are too optimistic.

6.3.5 Example

The structure of a graph corresponding to an implementation of register-indirect addressing has already been explained in section 6.2.7. We now demonstrate type propagation in this graph. As a starting point, we assume that the types returned by method getOp are RegAddr and Byte, and the types of the polymorphic register file RF are MemAddr, Byte, and Word, as determined by an analysis of other parts of the system description.

Figure 39 shows type propagation results in the microprocessor example, obtained as follows: The propagation of {RegAddr, Byte} to AR is constrained

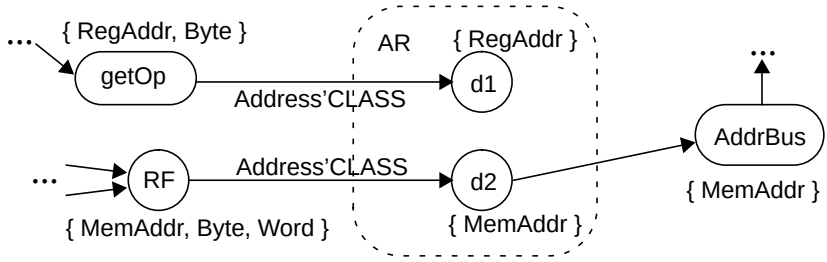


Fig. 39: Type propagation results

by Address'CLASS which includes RegAddr and MemAddr, but not Byte. Hence, after intersection, only RegAddr reaches d1. Similarly, of the possible types of RF only MemAddr is propagated to vertex d2. This type is further propagated (transitively) to the AddrBus vertex.

Distinguishing the two definitions of AR, d1 and d2, has prevented the type RegAddr from being propagated from AR to AddrBus. However, an implementation of AR must of course be able to represent the state of any class with which AR may be defined. This can be computed after FPI for every sequential object as the supremum over all its definitions. An analogous step is not necessary for concurrent objects whose definitions are all represented by a single vertex.

Results of an analysis of the complete microprocessor data flow are listed in figure 40. Table rows correspond to polymorphic objects, and columns to class types. An 'X' means that a polymorphic object must be able to contain a value of a class type. A '—' stands for a class type which can be excluded from synthesis thanks to data type analysis results. The blank fields result from types which are incompatible with the type constraint of the object and would not have to be considered anyway. Finally, bit-size values shown in the right-hand columns have been calculated from the analysis results according to equation 57. These values correspond to automatic optimization using the methods presented in this paper (auto), manual implementation of the 32-bit processor with bit-vector data types (man), and synthesis without any optimization (noopt).

Note that it has been necessary to provide external information about the memory contents to enable this analysis. This is because the memory contains user-defined data whose properties cannot be extracted from the microprocessor model. Assumptions about this data can be specified in a pseudo memory entity that is connected to the microprocessor for analysis. The

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

Fig. 40: Optimization results

memory entity contains some objects initialised so that their types and internal structure (e.g., polymorphic subobjects) reflect valid memory contents. By assigning these objects to the data bus, the type information represented is injected into the microprocessor model.

The main optimization potential in this case stems from the fact that instructions have only short operands (IR.OP) of class types Byte and RegAddr, but no full 32-bit words or memory addresses. This is recognised by analysis, allowing the reduction of IR and MEM width to 32 and 34 bits, respectively, instead of an unoptimized 80 bits. However, each memory element still has 2 bits more than a manually implemented 32 bit architecture. The reason are the tags used to distinguish values of the four class types present in MEM, requiring 2 bits when fully encoded (cf. section 4.4.2). While these tags are useful to detect run-time errors during simulation, e.g., loading of an operand into the instruction register, a manually designed hardware implementation of the microprocessor would in some cases make no difference between the encoded types. For instance, an operand would be reinterpreted as instruction when loaded into IR. On the other hand, the different instruction types definitely must be told apart by a tag (i.e., opcode). Techniques for automatic determination of cases that allow to save tags are the subject of the following section.

6.4 Determination of tag equivalence classes

The conditions under which tags can be encoded with the same value are elaborated in this section. We show how this information can be expressed as equivalent classes or partition of the tags, and how it is propagated though a system. By proving that this problem fulfils the axioms of a data flow analy-

sis framework, we show that the techniques presented above are applicable to the determination of tag equivalence classes, too.

6.4.1 Tags that make no difference

In order to determine tags that can possibly be saved, we first summarise what tags are needed for at all. As opposed to previous considerations from a language design point of view, the need for synthesis of tags may arise from:

1) The necessity for an aid in interpreting the structure of a polymorphic object's state. However, as we will see in the next chapter, this is unnecessary as particular data (i.e., an attribute) is always located at the same bit position in the synthesized state memory of any two classes related by inheritance.
2) Use of the TAG attribute in the source code. Without further investigation of the context of use, this makes it necessary to distinguish tags for all class memberships that a polymorphic object may have. The comparison with a constant tag, however, allows a less pessimistic analysis. Let P be a polymorphic object of C'CLASS. We distinguish the following cases:
   - P'TAG = Constant_Tag: The particular constant tag must be distinguishable from all other tags that correspond to C'CLASS.
   - P'TAG >= Constant_Tag: The tags of the class denoted by the constant tag and its derived classes must be distinguishable from the other tags of C'CLASS. However, there is no need to distinguish between any of the tags within one of these two groups.
   - P'TAG > Constant_Tag: We must be able to tell tags of classes that are derived from the class denoted by Constant_Tag apart from the set of Constant_Tag and all other tags for C'CLASS.
   - Similar considerations can be made for the /=, <=, and < operations.
3) Dynamic binding of a method invocation P.M(...) with a polymorphic object P declared with type C'CLASS. Then the tags for which different redefined versions of the method are called must be distinguishable. Tags for which the same implementation is invoked can be encoded with the same value.

According to these considerations, equivalence classes of tags that do not have to be distinguished can be defined for each use of a tag. We can represent these equivalence classes as a partition of the set of all tags or of their denoted class types. This partition is a collection of non-empty, disjoint sets whose union is the set of all class types. The types in one set can share the same tag value. Tags of types in different sets have to be encoded with differ-

ent values. One special set exists to collect types for which no tag value is needed at all. This set can be marked by including a special dummy element, say *void*, which we may add to *Types*.

Consider, for example, a set of types, *Types* = $\{t_1, t_2, t_3, t_4, t_5, t_6, void\}$, with an inheritance relationship and redefined methods, M and N, as shown in figure 41. For a polymorphic object declared with t_3'CLASS, one and the same implementation of M is invoked when it is a member of class t_3 or t_4. Another method implementation is invoked if the tag is t_5 or t_6. No encoding is needed for t_1 and t_2. This is reflected in the partition $\{\{t_3, t_4\}, \{t_5, t_6\}, \{t_1, t_2, void\}\}$. A similar consideration of a dynamically bound invocation of method N of leads to the partition $\{\{t_3, t_5, t_6\}, \{t_4\}, \{t_1, t_2, void\}\}$.

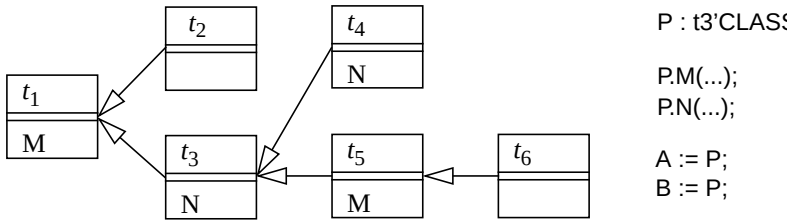


Fig. 41: Example for computation of tag equivalence classes

An encoding of P's tags must respect both these partitions. We must find a finer partition into equivalence classes so that the differentiation of class membership required for both method calls is possible. To minimize pessimism, this should be the coarsest possible partition. The order relation *finer* and a supremum operator are investigated in the subsequent sections. Intuitively, the pairwise intersection of both partitions' sets of equivalent tags, while filtering out empty sets, yields the desired result. In the above example: $\{\{t_3\}, \{t_4\}, \{t_5, t_6\}, \{t_1, t_2, void\}\}$.

6.4.2 Propagation of partitions

The existence of assignments makes it necessary to consider the propagation of tag partitions between objects. See, for instance, the assignments shown in figure 41. Any tags that are distinguished for the assignment target must be distinguished for the source, too. Otherwise it would not be possible after an execution of the assignment to make the distinction of tags that is necessary for the target object.

Note that in this case the flow of information is directed from the target of an assignment towards the source. In other words, it goes from the use of the source object (in the definition of the target) back to the source object's reaching definitions. Hence, the edges of a partition flow graph must be directed in the opposite direction as compared to section 6.2 so that the graph represents use-definition chains.

Over these chains, type partitions are propagated transitively. This, and the possible presence of cycles, necessitates the application of fixed-point iteration. While we have not implemented the determination of tag equivalence classes, we formulate the problem's solution as a data flow analysis framework and prove its well-formedness. This makes the techniques implemented for type analysis applicable to the type partitioning problem with a few modifications to be addressed.

6.4.3 Solution framework

We represent a partition as a subset of the power set $\wp(Types)$, i.e., as a set of subsets of *Types* for which the partition property holds. The set of all partitions, *Part*, is defined as follows:

$$Part = \{X \subseteq \wp(Types) \mid \bigcup_{A \in X} A = Types \wedge \forall A, B \in X : A \neq B \Rightarrow A \cap B = \emptyset \wedge \forall A \in X : A \neq \emptyset\} \tag{68}$$

A partition, $X \in Part$, is finer than a second partition, $Y \in Part$, if each element contained in X is a subset of some element in Y:

$$X \geq Y \equiv \forall A \in X \; \exists B \in Y : A \subseteq B . \tag{69}$$

The confluence of two partitions, X and Y, is defined by the pairwise intersection of the partition sets, without any empty sets. As we show in the following section, this is the coarsest partition that is finer than both A and B.

$$X \vee Y \equiv \{A \cap B \mid A \in X \wedge B \in Y \wedge A \cap B \neq \emptyset\} . \tag{70}$$

From this follows the top element as the finest partition and the bottom element as the coarsest one; the proof can be found in the next section.

Transfer functions are needed only to introduce knowledge by propagation via an edge from the entry vertex. For this purpose, we define a set of simple transfer functions, $F_{Part} = \{f_L : Part \to Part \mid L \in Part\}$, with:

$$f_L(X) := \begin{cases} X & if \;\; L = \top \\ L & if \;\; L \neq \top \end{cases} \tag{71}$$

The flow graph can be constructed similar to the description in section 6.2. The fixed-point iteration algorithm of section 6.3 can be reused with few changes, too. The following modifications are necessary:

- The direction of information propagation is reversed; information flows from the use of an object to its definition. Hence, edges that correspond to assignments are reversed, too. They are attributed with the identity transfer function, $f_{\top}$.
- Edges from the entry vertex, e, are attributed with $f_{\mu(e,d)}$, where $\mu(e,d)$ is tag partitioning information obtained from a dynamically bound method call or the use of the VHDL attribute TAG. Should there be several such uses of a single definition, $\mu(e,d)$ results from the confluence of this information.
- Top and bottom, transfer functions, and confluence operators as defined here are used in both, syntax-directed analysis and fixed point iteration.
- The order in which objects are processed during FPI is reversed. This helps to speed up convergence as the main information flow is now in inverse program order.

6.4.4 Proofs

To make sure that the fixed-point iteration algorithm presented in section 6.3.3 is applicable, we prove that the solution framework presented is a data flow analysis framework. We first show that the set of all partitions, *Part*, is closed under application of the proposed supremum operation:

Let $X, Y \in Part$. To be shown is $Z = X \vee Y \in Part$, i.e. the union of the sets in Z is *Types* and the subsets are pairwise disjoint and non-empty:

$$\begin{aligned} \bigcup_{C \in Z} Z &= \bigcup_{A \in X} \bigcup_{B \in Y} A \cap B = \bigcup_{A \in X} A \cap \bigcup_{B \in Y} B \\ &= \bigcup_{A \in X} A \cap Types = \bigcup_{A \in X} A = Types \end{aligned} \tag{72}$$

- Let $C_1 = A_1 \cap B_1 \in Z$ and $C_2 = A_2 \cap B_2 \in Z$. Let $C_1 \neq C_2$. It follows $\neg(A_1 = A_2 \wedge B_1 = B_2)$, which equals $A_1 \neq A_2$ or $B_1 \neq B_2$. As X and Y are partitions, this implies $A_1 \cap A_2 = \emptyset$ or $B_1 \cap B_2 = \emptyset$. Thus, $(A_1 \cap A_2) \cap (B_1 \cap B_2) = (A_1 \cap B_1) \cap (A_2 \cap B_2) = C_1 \cap C_2 = \emptyset$.
- $C \neq \emptyset$ for all $C \in Z$ follows directly from the definition of $\vee$.

Next to be shown is that the *finer* relation is a partial order:

- Reflexivity: Let $X \in Part$ and $A \in X$. It holds $A \subseteq A$. Hence, according to the definition of $\geq$ (equation 69): $A \geq A$. It follows $X \geq X$.

- Antisymmetry: Let $X, Y \in Part$ with $X \geq Y$ and $Y \geq X$. Let $A \in X$. Since $X \geq Y$, there exists $B \in Y$ such that $A \subseteq B$. Since $Y \geq X$, there exists $C \in X$ so that $B \subseteq C$. From transitivity of $\subseteq$ follows $A \subseteq C$. Hence, *A* and *C* are not disjoint. As *A* and *C* belong to the same partition, *X*, it must be $A = C$. Hence, $A \subseteq B \subseteq A$, i.e., $A = B$. It follows $X = Y$.
- Transitivity: Let $X, Y, Z \in Part$ with $X \geq Y$ and $Y \geq Z$. Let $A \in X$. Since $X \geq Y$, there exists $B \in Y$ such that $A \subseteq B$. Since $Y \geq Z$, there exists $C \in Z$ so that $B \subseteq C$. From transitivity of $\subseteq$ follows $A \subseteq C$. Hence, it holds $\forall A \in X\ \exists C \in Z : A \subseteq C$, i.e., $A \geq C$. It follows $X \geq Z$.

We now show that *Part* is a semilattice ordered by the *finer* relation. Every two elements of *Part* have to have a supremum. This is the previously introduced pairwise intersection of equivalence classes:

- An upper bound exists: Let $X, Y \in Part$. Let $U = \{A \cap B \mid A \in X \wedge B \in Y \wedge (A \cap B) \neq \emptyset\}$. Since $A \cap B \subseteq A$ and $A \cap B \subseteq B$, it follows $U \geq X$ and $U \geq Y$, i.e., *U* is an upper bound of *A* and *B*.
- *U* is the smallest upper bound: Let $U' \in Part$ with $U' \geq X$, $U' \geq Y$ be another upper bound of *X* and *Y*. Assume $\neg(U' \geq U)$. According to the definition, this means $\neg\forall C' \in U'\ \exists C \in U : C' \subseteq C$, which is equivalent to $\exists C' \in U'\ \forall C \in U : \neg C' \subseteq C$. Let $C' \in U'$ such that $\neg C' \subseteq C$ for all $C \in U$. Since $U' \geq X$ and $U' \geq Y$, there exist $A \in X$ and $B \in Y$ so that $C' \subseteq A$ and $C' \subseteq B$. Hence, $C' \subseteq A \cap B$. As $A \cap B \in U$, we have found a $C \in U$ with $C' \subseteq C$. This contradicts the assumption. Hence, any upper bound U' of *X* and *Y* must be greater (a finer partition) than or equal *U*.

A top and a bottom element exist:

- The bottom element is a partition of *Types* into only one set, *Types* itself: $\bot = \{Types\}$. Let $X \in Part$ and $A \in X$. The set $Types \in \bot$ fulfils $A \subseteq Types$. Hence, $X \geq \bot$.
- The top element is a partition of *Types* into sets each of which contains only a single type: $\top = \{\{t\} \mid t \in Types\}$. Let $A = \{t\} \in \top$. Let $X \in Part$. Let $B \in X$ be the unique element of partition *X* which includes *t*. As *t* is the only element in *A*, $A \subseteq B$ follows immediately. Hence, $\top \geq X$.

Finally, properties of the transfer functions are proved:

- Monotony: Let $f_L \in F_{Part}$, $X, Y \in Part$, and $X \geq Y$. If $L = \top$, $f_L(X) = X$ and $f_L(Y) = Y$. Hence, $f_L(X) \geq f_L(Y)$. If $L \neq \top$, $f_L(X) = L$ and $f_L(X) = L$. Again, $f_L(X) \geq f_L(Y)$.
- Identity function: $f_\top(X) = X$ for all $X \in Part$.

- Closedness under composition: Let $f_L, f_M \in F_{Part}$. If $L = \top$, $f_L(f_M(X)) = f_M(X)$. If $L \neq \top$, $f_L(f_M(X)) = L = f_L(X)$. Hence, the composition of f_L and f_M is in F_{Part}.
- Let $L \in Part$. For $f_L \in F_{Part}$ holds $f_L(\top) = L$.
- Distributivity: Let $f_L \in F_{Part}$ and $X, Y \in Part$. If $L = \top$ then $f_L(X \vee Y) = X \vee Y = f_L(X) \vee f_L(Y)$. If $L \neq \top$ then $f_L(X \vee Y) = L = L \vee L = f_L(X) \vee f_L(Y)$.

Given all these properties, we conclude that the solution framework proposed for finding tags that do not need to be distinguished is a distributive data flow analysis framework. Hence, our fixed-point iteration implementation developed for type propagation is as well applicable to the new problem with the changes described in the previous section.

6.5 Summary

We have presented data type analysis techniques targeted at an optimized synthesis of polymorphic objects and dynamic binding. Analysis is carried out statically with a combination of data flow analysis techniques for constructing definition-use and use-definition graphs, respectively, and a fixed-point iteration for propagating type information. Its results can be used to implement objects with a minimized number of bits and to simplify the dynamic binding of redefined methods.

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

Experimental results show a significant benefit of our techniques as compared to non-optimized synthesis of object-oriented data types. Currently, we cannot fully achieve the results one would obtain from a manual design at bit level, i.e., without data abstraction. However, as the treatment of the type partitioning problem shows, data flow analysis can be exploited to utilise further optimization potential.

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

Chapter 7

VHDL Code Generation

After analysing and optimizing an object-oriented model, synthesis steps must be performed in order to obtain a digital circuit implementation. This includes the synthesis of the method algorithms, of storage for an object's state, and of inter-object communication. It is the aim of this work not to re-implement synthesis algorithms. Instead, as much existing technology as possible shall be used so as to be able to focus on the new, object-oriented aspects. To this end, we target the generation of VHDL code for further processing with third-party tools.

Alternatives for the translation of an object-oriented model into synthesizable VHDL are discussed in the following section. After making a choice, we present how a bit-level encoding of object data can be represented in VHDL. Next, we address the transformation of methods into synthesizable VHDL code. Another section describes VHDL templates for implementing object behaviour, i.e., the server's invocation of methods in response to incoming messages and the actions taken by a client to send a message. Finally, the instantiation and interconnection of objects is transformed into a VHDL model.

7.1 Translation alternatives

In order to discuss alternatives for the translation of object-oriented models into VHDL, we first point out the goals of code generation. Particularly, the focus on synthesis results in requirements different from related, simulation-oriented work [145]. We identify procedural (behavioural) and structural translation as the main alternatives (cf. section 3.3.1), point out their differences, and discuss their pros and cons. The structural approach will be followed in the remainder of this chapter.

7.1.1 Design goals

The primary goal of this chapter is to describe the generation of VHDL code from which the circuit structures of section 4.4 can be synthesized. That is, the VHDL code does not necessarily have to describe these structures directly; they may be generated by a VHDL synthesis tool. This allows to delegate to existing tools a significant portion of the overall task of generating hardware from an object-oriented model. Thereby, we can validate the concepts of this work without implementing a complete synthesis tool.

When generating VHDL code, we must be aware of the different description styles which correspond to different levels of abstraction and for which different tools exist. Register transfer level (RTL) synthesis tools accept descriptions consisting of finite state machines and datapaths, where operations are bound to specific states or clock cycles. Hence, if we chose to generate RTL style VHDL, we would have to implement as part of the object synthesizer the synthesis of a data path and controller that implement the methods' algorithms.

This complex task can be performed by a high-level synthesis (HLS) tool. However, we must make sure that VHDL code generation complies with the synthesis tools' requirements and restrictions. Whereas a VHDL subset compatible with all major RTL synthesis tools is well-documented [75] and understood, different HLS tools require rather different coding styles and language subsets. This makes it necessary to tailor the generated code for a specific HLS back-end. For this work, the Synopsys Behavioral Compiler (BC) has been chosen for its availability in the Europractice programme and due to project partners' preferences. Academic tools have not been considered, although they may implement more advanced synthesis and optimization concepts, because of concerns regarding their maturity and their degree of VHDL support.

In addition to synthesizability of the generated VHDL code, we must ensure a wide support for the object-oriented models that are fed into our tool. Inheritance and polymorphism should be supported as well as generic modelling, and must be transformed into synthesizable VHDL. Furthermore, the generated code should be as concise and readable as possible, and enable the user to correlate it with the original model for simulation and debugging purposes. In the following, alternatives are discussed with these criteria in mind.

7.1.2 Generation of procedural code

A translation of object-oriented VHDL, making extensive use of VHDL's packages, subprograms, and composite data types, has been proposed in [145]. It can be summarised and adapted to our terminology as follows:

- From a class type, a record with elements corresponding to the class's attributes is created. A derived class type is translated into a record which includes the inherited attributes, too. A class-wide type results in a record which collects not only the attributes of its base type, but also any attribute declared in a derived class. In addition, the record has an element for storing a type tag.
- Methods are translated into subprograms with a formal parameter that denotes the object with which the method is invoked. The access to attributes is translated into an access of the corresponding record elements of the object parameter.
- All declarations resulting from the translation of a class are collected in a VHDL package.
- In the declaration of an object, a class type or class-wide type is replaced by the corresponding record.
- The invocation of a method with an object is translated into a subprogram call, and the object is passed as actual parameter to the method. Dynamic binding, if necessary, is performed by a case statement that invokes the redefined versions of a method depending on a polymorphic object's tag value.
- Type conversions are introduced in assignments between objects of compatible class types (cf. section 5.3.3). This is necessary as the different record types resulting from translation are no longer compatible for assignment.

Indeed, such a translation has been implemented for Objective VHDL in order to enable a simulation on VHDL level. Its advantages are the readability of the record representation of attributes and the similarity of the original and generated code. Beyond syntactical similarity, it must be emphasized that the translation does not affect the timing behaviour of the model. No delays, not even delta delays, are introduced or removed.

However, there are several drawbacks to this approach:

- VHDL does not provide parameterization of packages nor records. Hence, the translation of generic class types is difficult. Only if the value of the generic is known, a record translation can be created. This is not always the case as translation is performed before the elaboration (cf.

section 5.5) which may be required to compute generic values. Furthermore, a new record has to be created for each different generic value that occurs in a model.

- A package can only have a single body. This makes it hard to provide multiple versions of a class's implementation. The need for such a diversified translation arises from the optimization of individual objects, e.g. based on the analyses presented in the previous chapter. For instance, a method that is never invoked with some object need not be implemented for this object, but it has to be provided for the other objects of the class.

Further problems arise when we take synthesis aspects into consideration:

- The VHDL HLS tool may be unable to handle the description efficiently. Particularly, it may have to synthesize every single object and every single method invocation from the source code rather than by inferring a library component. This disables the use of pre-synthesized classes or classes whose source code is not supplied (e.g., IP). These considerations hold, in particular, for the HLS tool used in this work. While BC provides so-called preserved subprograms, scheduled subprograms, and the map-to-operator pragma to avoid the inlining of subprograms [152], all these features have restrictions which make it in general impossible to apply them to the subprograms that are created from methods.
- The translation does not allow an easy integration of conditional request acceptance and arbitration of concurrent clients. Particularly, as discussed in section 3.1.5, the modelling style for guarded method invocation presented in [145] is not synthesizable.
- The translation of class data into records is problematic because of tool limitations. While most synthesis tools support records, support for record aggregates is not common [75]. Furthermore, nested composite types, e.g. a record element which is again a record, may cause trouble. Such data structures, however, are the result of translating class composition.
- Finally, translation of class-wide types into records would typically cause significant hardware overhead. This is because the record resulting from translation includes an individual element for each attribute of the base class and *all* its derived classes whereas only the attributes of *one* specific class are needed at a time. A better implementation would share storage among classes from different branches of the inheritance hierarchy. This, however, requires to map attributes to a bit-level representation. For instance, a translation of Buffer'CLASS into a record would look as follows:

```
type Buffer_CLASS is record              -- (size and bits are parameters)
    item :   Buffer_array;               -- size * bits      (from Buffer)
    first :  Integer range 0 to size – 1; -- log2(size)      (from FIFO)
    nxt :    Integer range 0 to size – 1; -- log2(size)      (from FIFO)
    empty :Boolean;                      -- 1                (from FIFO)
    index : Integer range 0 to size;     -- log2(size+1)     (from LIFO)
end record;                              -- total #bits: sum of the above
```

The element item is needed for both FIFO and LIFO. However, the other elements are needed only for either FIFO or LIFO. The storage of first, nxt, and empty on the one hand and index on the other hand could be shared because a polymorphic object is a member of only one class at a time. However, VHDL's type system does not provide the means, e.g. variant records or unions, to do so. Details of a bit-level encoding that allows to share attribute storage are presented in section 7.2.

7.1.3 Generation of structural code

Structural translation creates VHDL design entities (entity-architecture pairs) as the implementation of classes. This has several advantages over procedural translation:

- Physical connections and components required for the implementation of message passing can be described directly instead of relying on BC's implementation of subprogram calls. This enables us to infer communication and arbitration implementations during object synthesis.
- The object synthesizer can make use of pre-synthesized components by simply instantiating them.
- The generic parameters available with entities (as opposed to packages) can be utilised in the translation of parameterized classes.
- Multiple architectures of an entity allow to describe different optimized implementations of a class and to handle them using VHDL features.

Hence, many of the major problems of the procedural approach can be tackled by structural translation. However, there are some disadvantages:

- The deviation from the original code is larger, making comprehension of simulation results harder.
- A translation of method invocation into signal communication necessarily creates additional delta delay in the VHDL model compared to the potentially immediate method execution of the object-oriented model. Thus, structural translation does not preserve semantics with respect to detailed timing.

These issues are acceptable when aiming at synthesis rather than simulation. A synthesized design is at a lower level of abstraction than the specification from which it has been created, and is therefore necessarily dissimilar in terms of description style. Assuming a synchronous design style, values that occur in the specification model are relevant only at special points in time, e.g., at the rising edge of a clock signal. Synthesis tools typically ignore anything that happens at a finer granularity of time, including delta delays. Hence, we choose to follow a structural translation approach in this work.

7.2 Encoding of object data

The first step towards a VHDL representation of an object-oriented synthesis model is to express the encoding of an object's state space (cf. section 4.4.1 and 4.4.2) in VHDL. To this end, we present a bit-vector (std_logic_vector) representation of VHDL's scalar and composite types, which may be used as the type of object-oriented attributes, while leaving non-synthesizable access and file types out of consideration. With this basis, the encoding of Objective VHDL's object-oriented types, namely class types, derived class types, and class-wide (polymorphic) types, is described.

For each bit-vector representation, we define a synthesizable conversion from and into the original type. This will allow us to adapt methods to the encoded data representation without having to synthesize their functionality to the bit level.

7.2.1 Scalar types

Scalar types are enumeration, integer, real, and physical types (in VHDL). Similar types may be pre-defined as part of an object-oriented language, or user-defined using a language's type definition mechanisms. We exclude from our considerations the real and physical types because they are not supported by synthesis tools [75]. Synthesizable variants of these types can be provided in a library or implemented by the user as class types; an example of a fixed-point type will be presented later in section 8.4.1.

We first consider the encoding of enumeration types. This includes a definition of the their encoding length b and their encoding function *enc* (cf. section 4.4.1). Moreover, we show how conversions from the enumeration type into the encoded representation and vice versa are provided. Respective conversion functions are supplied in the package Syn_Scalar_Types for all pre-defined or standardised enumeration types (see appendix E).

- Std_(u)logic: The mapping of std_ulogic and std_logic (resolved subtype of std_ulogic) values to a std_logic representation is trivially performed by identity:

(73) $b(\text{Stdl}) = 1\,,\ enc_{\text{Stdl}}(v) = v\,.$

- Bit: The values of data type bit, '0' and '1', are encoded by their std_logic counterparts. The functions Bit_to_Stdl(Bit) return Std_logic and Stdl_to_Bit(Std_logic) return Bit are provided for conversion.

(74) $b(\text{Bit}) = 1\,,\ enc_{\text{Bit}}(v) = v\,.$

- Boolean: Deliberately choosing active-high logic, the boolean value true is encoded as '1', and false as '0'. Conversions are performed by the functions Bool_to_Stdl(Boolean) return Std_logic and Stdl_to_Bool(Std_logic) return Boolean.

(75) $b(\text{Bool}) = 1\,,\ enc_{\text{Bool}}(\text{true}) = \text{'1'}\,,\ enc_{\text{Bool}}(\text{false}) = \text{'0'}\,.$

- Other pre-defined enumeration types are character, severity_level, file_open_kind, and file_open_status. Their encoding and synthesis is not implemented. It might make sense only for characters. This could be dealt with in a way similar to user-defined enumeration types.

User-defined enumeration types are specified by listing all their enumeration literals. Their encoding could be performed by obtaining the positional number of a literal with the VHDL attribute POS and encoding this value as an integer. However, POS is not supported for VHDL synthesis [75]. This makes it necessary to generate conversion functions for each user-defined enumeration type <ud_enum>: <ud_enum>_to_Stdv(<ud_enum>) return Std_logic_vector and Stdv_to_<ud_enum>(Std_logic_vector) return <ud_enum>. These functions can be inserted immediately after the declaration of the enumeration type. They are implemented with case statements that return the encoded value for each enumeration literal and vice versa. In this work, binary encoding is used, which leads to equation 76 for encoding length. Other encodings, e.g. one-hot, could be added easily. A user-defined choice could be specified like in RTL synthesis with the VHDL attribute ENUM_ENCODING.

(76) $b(\text{<ud\_enum>}) = \lceil \log_2(\#\text{literals}) \rceil\,,\ enc_{\text{<ud\_enum>}}(v) = pos(v)_{(2)}$

Integer types include the pre-defined type Integer and user-defined types which specify a range of integer values. Furthermore, there are pre- and user-defined subtypes with restricted range. All these can be characterised by the lower bound *lb* and the upper bound *ub* of legal values, and an encoding can be defined as follows:

- Non-negative integers: If the lower bound *lb* is non-negative, an unsigned binary encoding of natural numbers is used. This encoding represents the values from 0 up to the upper bound *ub* with the number of bits defined in equation 77. For conversion, the functions Nat_to_Stdv(Natural, Natural) return Std_logic_vector and Stdv_to_Nat(Std_logic_vector) return Natural are supplied. Note that the second parameter of Nat_to_Stdv specifies the number of bits used in the encoding. For Stdv_to_Nat, this can be derived from the index range of the std_logic_vector that is passed as parameter. The synthesizable implementation of these conversions is based on the IEEE numeric synthesis package or, for older versions of Synopsys, the STD_LOGIC_ARITH package.

(77) $b(\text{Natural}) = \lceil \log_2(ub + 1) \rceil,\ enc_{Nat}(v) = v_{(2)}$

- Integer (sub)types that include negative numbers: If the lower bound *lb* is negative, a twos-complement encoding of integer numbers is used. The respective conversion functions are Int_to_Stdv(Integer, Natural) return Std_logic_vector and Stdv_to_Int(Std_logic_vector) return Integer. The encoding bit-width (equation 78) can be derived from the value range of *b*-bit twos-complement, which is $-2^{b-1}, \ldots, 2^{b-1} - 1$.

(78) $b(\text{Integer}) = \lceil \log_2(max(|lb|, |ub| + 1)) \rceil + 1,\ enc_{\text{Int}}(v) = v_{(2)}.$

- User-defined integer types, available in Objective VHDL, are closely related to the pre-defined integer type. Their encoding is defined in analogy. By adding an explicit type conversion into (and from) the pre-defined integer types, they can be converted to (and from) std_logic_vector using the functions mentioned above. The explicit type conversion is available in VHDL.

The bounds, *ub* and *lb*, may be parameterized with generic values that are not yet known. In Objective VHDL, this holds for subtypes. See, for instance, the attributes first, nxt, and index in the buffer example (section 4.1.9). Parameterized bounds make it necessary to have encoding width computation done by VHDL elaboration, during which the generics become known[1]. This is achieved by inserting expressions corresponding to equation 76–78 into the VHDL code instead of pre-computed values. For this purpose, the functions max(Integer_list) return Integer and intlog2(Natural) return Natural are provided in the Syn_Scalar_Types package. Max returns the maximum of its

1. The deeper reason for this is that the available Objective VHDL tool world provides no elaboration so that the object synthesizer must rely on an analysed, pre-elaboration model (cf. section 5.5).

arguments and intlog2 computes the function $\lceil \log_2(.) \rceil$. The implementations of these functions can be evaluated by the elaborator. They are not synthesized into hardware.

7.2.2 Composite types

Composite types in (Objective) VHDL, as in most programming languages, are records and arrays. Records are a heterogeneous collection of elements whereas arrays are a homogeneous collection of elements. The encoding of these composite types and their conversion from and into a bit level representation can be defined based on the elements' encoding and conversion. In addition, the access to individual elements must be enabled. Respective techniques are addressed in the remainder of this section. Their implementation is provided in the package Syn_Composite_Types (see appendix E).

A record type is characterised by its elements and their types. Let there be a record *R* with *n* elements of types $t_1, \ldots, t_n$. Its encoding length is defined by the sum of the element types' encoding lengths. The encoding can be defined as the concatenation of the encodings of the elements' values $v_1, \ldots, v_n$. This is similar to class types (cf. section 4.4.1).

(79) $b(R) = b(t_1) + \ldots + b(t_n)$

(80) $enc_R(v_1, \ldots, v_n) = (enc_{t_1}(v_1), \ldots, enc_{t_n}(v_n))$

Access to the individual elements must be expressed in the generated VHDL. For this purpose, it is necessary to know the start position $pos_R(i)$ of the *i*-th element in the encoded bit string. The first element's encoded value may start at $pos_R(1) = 0$ (any index offset could be chosen) and requires $b(t_1)$ bits. Hence, the second element can start at $pos_R(2) = pos_R(1) + b(t_1)$. More generally, $pos_R(i+1) = pos_R(i) + b(t_{i+1})$ or, without recursion:

(81) $$pos_R(i) = pos_R(1) + \sum_{j=1}^{i-1} b(t_j),\ i \in \{1, \ldots, n+1\}.$$

VHDL code is generated so that these values are stored in a constant array R_pos declared with type Param_Vector which is pre-defined in package Syn_Composite_Types. It is also possible to inline the position expressions wherever they are needed; however, this makes the code less readable. This holds particularly if $pos_R(j)$ depends on generic parameters so that it cannot be computed by the object synthesizer.

Obviously, the end position of an element is one less than the start position of the next element. Note that the case of $i = n + 1$ in equation 81 is needed to define the end position of the last element this way. Having the element position values, an element of an encoded record instance can be accessed by a VHDL slice:

```
record_instance( R_pos(i) to R_pos(i+1) – 1 )
```

This slice can occur on the left hand of an assignment as well as on the right hand (in an expression). Hence, it allows both to assign and to read an element. In the first case, any required conversion into the encoded representation can be added to the assigned expression. In the second case, the slice itself can be converted. Note that, as required by VHDL, in both cases a right-hand value is converted.

Our considerations of array types are limited to one-dimensional arrays for the time being. This restriction is lifted later. A one-dimensional array *A* is characterised by the lower and upper bound, *lb* and *ub*, of its index range, and its element type *t*. Its number of elements is $n = ub - lb + 1$. The number of bits required for its encoding is *n* times the size of a single element. The encoding of an array value is defined as the concatenation of the encodings of the element values:

(82) $b(A) = n \cdot b(t),\ enc_A(v_{lb}, \ldots, v_{ub}) = (enc_t(v_{lb}), \ldots, enc_t(v_{ub}))$.

Given the start position of the lowest-index element, $pos_A(lb)$, the start position of element number $i \in \{lb, \ldots, ub + 1\}$ is at:

(83) $pos_A(i) = pos_A(lb) + (i - lb) \cdot b(t)$

Similar to records, the case of $i = ub + 1$ is required to obtain the end position of element *i* as $pos_A(i + 1) - 1$. However, it must be emphasized that *i*, the index with which an array element is addressed, can be a variable whereas the *i* in equation 81 is a constant corresponding to a specific record element. This difference is important as a slice with variable range is not synthesizable [75]. Hence, a different, synthesizable solution for read and write access to a particular element has to be found.

There are two known workarounds. These are the implementation of bitwise access controlled by a for loop that iterates over the range and, second, the use of a case statement that selects different constant ranges depending on *i*. The first solution works for RTL synthesis but not with BC since this tool cannot unroll for loops during elaboration. The other solution cannot be used if the number of elements is parametric. Hence, a novel approach had to be developed.

Our implementation of array element access is based on describing a multiplexer hierarchy for element selection. This structure, shown in figure 42, first selects the left or right half of the complete array according to the value of the most significant bit (MSB) of the index. In further stages, bisection is continued until finally a single element can be selected depending on the least significant index bit (LSB).

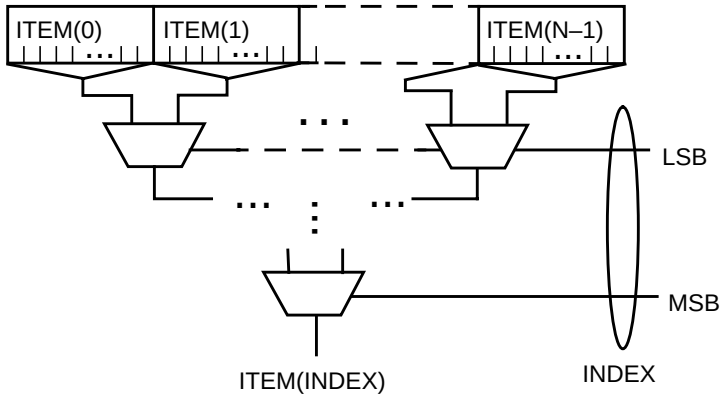


Fig. 42: Multiplexer hierarchy for array element selection

The description of this hierarchy must integrate with the algorithmic code of the methods, in which attributes are accessed. Hence, it must be modelled in a behavioural way, but not as a structure. To this end, we provide a function, Select_Element, that is synthesized into the desired structure. Its parameters are the following:

- A is an std_logic_vector that stores the encoded value of an array.
- INDEX is the index i of the element to be selected.
- SIZE is the size $b(t)$ (in bits) of a single element.
- BASE is the lower bound lb of the range of legal values of INDEX.

The function implements the same principle as the multiplexer hierarchy: It splits the complete array at its pivotal element, selects the half that contains the indexed element, and performs the same operation with this half. Recursion stops when exactly the slice holding the value of the indexed element is selected and can be returned.

```
function Select_Element(
            constant A : Std_logic_vector;
            constant INDEX, SIZE, BASE : Natural )
```

```
                return Std_logic_vector is
    constant PIVOT : Natural := ((A'RIGHT – A'LEFT + 1) / SIZE) / 2;
begin
    if A'RIGHT – A'LEFT + 1 = SIZE then  -- end of recursion
        return A;
    elsif INDEX < BASE + PIVOT then      -- left half of array
        return Select_Element(
                    A( A'LEFT to A'LEFT + PIVOT * SIZE – 1 ),
                    INDEX, SIZE, BASE );

    else -- INDEX >= BASE + PIVOT        -- right half of array
        return Select_Element(
                    A( A'LEFT + PIVOT * SIZE to A'RIGHT ),
                    INDEX, SIZE, BASE + PIVOT );
    end if;
end;
```

A similar procedure, Assign_Element, implements the assignment of an array element. The array is declared as a variable of mode **inout** and the value passed as an additional parameter compared to Select_Element.

```
procedure Assign_Element(
                variable A : inout Std_logic_vector;
                constant INDEX, SIZE, BASE : Natural;
                constant VALUE : Std_logic_vector );
    constant PIVOT : Natural := ((A'RIGHT – A'LEFT + 1) / SIZE) / 2;
begin
    if A'RIGHT – A'LEFT + 1 = SIZE then  -- end of recursion
        A := VALUE;
    elsif INDEX < BASE + PIVOT then      -- left half of array
        Assign_Element(
                    A( A'LEFT to A'LEFT + PIVOT * SIZE – 1 ),
                    INDEX, SIZE, BASE );

    else -- INDEX >= BASE + PIVOT        -- right half of array
        Assign_Element(
                    A( A'LEFT + PIVOT * SIZE to A'RIGHT ),
                    INDEX, SIZE, BASE + PIVOT );
    end if;
end;
```

Both these subprograms work correctly even if the number of array elements is not a power of two. It is important to note that they can be elaborated by a synthesis tool. In particular, recursion can be resolved during this process

because the recursion depth depends only on parameters with statically known values, but not the index. RTL synthesis of the subprograms creates the same result as direct synthesis of access of an array not encoded to the bit level. However, the HLS tool used (BC) attempts to schedule the comparison operations. This not only takes a lot of time, but has also lead to incorrect implementations in some experiments. Attempts to prevent the scheduling using BC's preserved subprograms, scheduled subprograms, mapping to a pre-synthesized module, and RTL synthesis pragmas have all failed due to tool limitations. Hence, alternative implementations based on bit comparisons, which are not subject of scheduling, have been developed and are provided in the Syn_Composite_Types package (see appendix E).

Considerations up to now only suffice for arrays of scalar elements. If the elements are again of a composite type, further actions are necessary to access the inner elements:

- An offset parameter is added to the access subprograms in order to support arrays of records. After selecting a record-typed element of an array, the offset is used for selecting a particular element of this record.
- The modification of the index, size, base, and offset parameters to be arrays of values rather than single values allows to deal with nested arrays. After completing the selection of an element according to the first index, Select_Element and Assign_Element are invoked recursively to perform further element selection until all indexes have been processed. The use of the offset parameter enables an arbitrary mix of array and record element selection.
- Multidimensional arrays can be implemented like arrays of arrays. This does not affect their language semantics, which differs from nested arrays in some respect in VHDL.

Thereby, all VHDL composite types can be mapped to a bit level representation in a synthesizable manner, regardless of their complexity and depth of nesting. The resulting interfaces are listed below. Their full implementation can be found in appendix E.

```
type Param_vector is array( Positive range <> ) of Natural;

function Select_Element(
        constant A : Std_logic_vector;
        constant INDEX, SIZE, BASE, OFFSET : Param_vector )
        return Std_logic_vector;

procedure Assign_Element(
        variable A : inout Std_logic_vector;
```

```
constant INDEX, SIZE, BASE, OFFSET : Param_vector;
constant VALUE : Std_logic_vector );
```

7.2.3 Object-oriented types

Object-oriented types include classes, derived classes, and class-wide types (of the polymorphic objects). Their encoding aspects have already been addressed in section 4.4.1 and section 4.4.2. We now focus on their implementation by VHDL code generation.

The encoding of a non-derived class *C* is very similar to a record's encoding if we relate class attributes to record elements. Hence, the principles developed in section 7.2.2 are applied to the VHDL bit level representation of class data and the access of individual attributes. Let $t_{C,1}, \ldots, t_{C,n(C)}$ be the types of *C*'s attributes as defined in section 4.1.2. The start position of the *i*-th attribute's value is

$$pos_C(i) = pos_C(1) + \sum_{j=1}^{i-1} b(t_{C,j}),\ i \in \{1, \ldots, n(C)+1\}, \tag{84}$$

where again $pos_C(1)$ can be chosen and $pos_C(n(C)+1)$ allows to define the end position of the last attribute as $pos_C(n(C)+1)-1$.

As we know from chapter 4, a derived class *D* extends its parent class *P*. This is reflected in its encoding that has been defined in section 4.1.4. The position of the inherited attributes in a bit-level representation of *D*'s data is chosen the same as for *P*. The additional attributes of *D*, numbered $1, \ldots, n(D)$ and being of types $t_{D,1}, \ldots, t_{D,n(D)}$, follow thereafter:

$$pos_D(1) = pos_P(n(P)+1) \text{ and} \tag{85}$$

$$pos_D(i) = pos_D(1) + \sum_{j=1}^{i-1} b(t_{D,j}),\ i \in \{1, \ldots, n(D)+1\}. \tag{86}$$

In order to encode a polymorphic object of class-wide type R_{poly}, we follow equation 39 and equation 41 of section 4.4.2 and take the results of data flow analysis (cf. section 6.4) into account. The task can be divided into the encoding of tags and of attributes. The required number of different tag values equals the number of tag equivalence classes minus one (the equivalence class of void tags). The number of tag bits depends on the encoding chosen, too. While different binary encodings such as one-hot could be chosen, we have implemented VHDL code generation for a binary encoding. Hence,

(87) $b_{Tag}(obj) = \lceil \log_2(|T_{eq}(obj)| - 1) \rceil, obj \in R_{poly}.$

An std_logic_vector subtype, <R>_ALL_TAGS, with this number of bits is created as well as tag constants that represent the tags' encoded values. Note that this is object-specific in that different objects of the same class-wide type may have different tag equivalence classes and therefore different tag encodings. These will be represented by different architectures of an object entity (see section 7.4.2).

For the encoding of polymorphic object data, it is important to emphasize that the same attribute can be found at the same start and end location in the encoding of the class in which it is declared and all derived classes. This only requires to choose the same $pos_C(0)$ (where *C* is the root of an inheritance tree) for all of them. As we place the tag at the beginning of the bit string, we choose $pos_{Tag} = 0$. The class data starts after the tag. Hence, $pos_C(0) = b_{Tag}(obj)$.

The position of an attribute can now be determined according to the *pos* values that have been defined above for non-derived and derived classes. This enables us to use a method defined for some class *C* also with an object of any derived class *D* as well as a polymorphic object of any of these classes. As opposed to a translation approach based on creating records for class data, methods do not have to be re-translated for every single derived class and class-wide type.

The total number of bits for encoding the polymorphic objects follows according to equation 41 as the sum of the tag encoding size and the maximum of the sizes of derived classes. However, only those classes whose membership is actually taken on by the object (cf. section 6.3) need to be taken into account:

(88) $b(obj) = b_{Tag}(obj) + max\{b(C) \mid obj \text{ takes on } C \text{ membership}\}$

7.2.4 Example

The following code excerpts present the VHDL code generated for the bit level representation of data of the buffer example (cf. section 4.1.9, section 5.2.4).

No translation is generated for class type Buffer_t as it is abstract. Only the supplementary declarations (those declarations which are no attributes and methods) are maintained because they might be used elsewhere in the code. These are the subtype Item_t and the type Buffer_array. We have put the width of their encoding as a comment:

```
subtype Item_t is Integer range 0 to 2**bits – 1;       -- bits
type Buffer_array is array( 0 to size – 1 ) of Item_t;  -- size * bits
```

It should be mentioned that bits and size, the generics of Buffer_t, are translated into entity generics as we will see in section 7.4.1.

The positions of the attributes of the derived class type FIFO are defined in a constant array, FIFO_POS, as listed below. For practical reasons, we also collect the positions of inherited attributes in the position array of a derived class. Furthermore, the position of the very first attribute is normalised to zero. An adaptation to a possible tag offset will be done in the methods. Finally, the FIFO's total size is defined as a constant, FIFO_SIZE, and a correspondingly sized std_logic_vector subtype, SYN_FIFO, is declared.

```
constant FIFO_POS : Param_vector :=
    (   0,
            -- item : Buffer_array;                  -- size * bits
        size * bits,
            -- first : Natural range 0 to size – 1;  -- intlog2(size)
        size * bits + intlog2(size),
            -- nxt : Natural range 0 to size – 1;    -- intlog2(size)
        size * bits + intlog2(size) + intlog2(size),
            -- empty : Boolean;                      -- 1
        size * bits + intlog2(size) + intlog2(size) + 1 );
constant FIFO_SIZE : Natural := FIFO_POS(4);
subtype SYN_FIFO is Std_logic_vector( 0 to FIFO_SIZE – 1 );
```

An analogous translation is generated for class LIFO:

```
constant LIFO_POS : Param_vector :=
    (   0,
            -- item : Buffer_array;                  -- size * bits
        size * bits,
            -- index : Natural range 0 to size       -- intlog2(size + 1)
        size * bits + intlog2(size + 1) );
constant LIFO_SIZE : Natural := LIFO_POS(2);
subtype SYN_LIFO is Std_logic_vector( 0 to LIFO_SIZE – 1 );
```

To create the translation of an object of the class-wide type Buffer_t'CLASS, let us assume that this object can be a member of class FIFO and LIFO. For the tag encoding of these two classes, $b_{Tag}(obj) = \lceil \log_2(2) \rceil = 1$ bit is needed. Hence, the tag subtype, SYN_BUFFER_ALL_TAGS, is generated as an std_logic_vector with one element. Tag constants of this subtype,

SYN_FIFO_TAG and SYN_LIFO_TAG, are defined with the tag values “0“ and “1“.

The object’s encoding size, stored in constant BUFFER_CLASS_SIZE, is the sum of the tag length, i.e. 1, and the maximum of the FIFO’s and LIFO’s sizes. This value cannot be computed by the object synthesizer as FIFO_SIZE and LIFO_SIZE depend on the generic parameters size and bits (see above) whose values are unknown before VHDL elaboration. Hence, an expression to be evaluated by the VHDL tool is inserted. Finally, an std_logic_vector subtype of the required size is declared.

```
subtype SYN_BUFFER_ALL_TAGS is std_logic_vector(0 to 0);
constant SYN_FIFO_TAG : SYN_BUFFER_ALL_TAGS := “0“;
constant SYN_LIFO_TAG : SYN_BUFFER_ALL_TAGS := “1“;
constant BUFFER_CLASS_SIZE : Natural :=
                1 + intmax( (FIFO_SIZE, LIFO_SIZE) );
subtype SYN_BUFFER_CLASS is Std_logic_vector( 0 to
                BUFFER_CLASS_SIZE – 1 );
```

7.3 Translation of methods

In section 4.1.3, we have characterised a method’s interface by the method name, its parameter names, the parameters’ (and eventually return value) types and directions (input, output, or both). Declaration and implementation of methods using Objective VHDL have been demonstrated in section 5.2.4. Given this model information, we will now show the generation of equivalent, synthesizable VHDL subprograms from which a subsequent HLS step can create the implementation of the method blocks shown in figure 17 and 18 of section 4.4.

7.3.1 Method interface

From a method interface as characterised in section 4.1.3, a VHDL subprogram declaration with the following properties is generated:

- The subprogram is a function if the method has a return value, all its parameters are input parameters, and it does not modify any attribute. Otherwise, it is a procedure. This is necessary because a VHDL function must have input parameters only. While Objective VHDL is defined with restrictions so that a method that returns a value can always be translated into a VHDL function, this is not true for other languages, e.g., C++.

- The subprogram name is the same as the method's name. Only if this would be illegal in VHDL, renaming is performed. This may be necessary if an OO language permits the use of names that are not allowed in VHDL. If Objective VHDL is used as input language, the only reason for renaming is a conflict with another declaration that uses the same name in the generated VHDL code.
- The parameters of the method are translated into parameters of the VHDL subprogram. This involves dealing with their types, storage classes, and directions as follows:
  - In general, the parameters must be encoded using the techniques described in section 7.2. In this case, the parameter type becomes a constrained std_logic_vector subtype. However, if a method parameter has a type to which a VHDL type corresponds, this VHDL type can be used. This is trivially the case if an OO model is described in Objective VHDL. Leaving the original types in place allows us to let the VHDL elaborator narrow down the value range and number of bits of the parameter by examining the actual parameters associated with it.
  - The VHDL mode **in** is used for input-only method parameters. Output-only ones become **out** parameters of the VHDL subprogram. Bidirectional method parameters (input and output) are translated as **inout**.
  - The VHDL storage class is **constant** for input-only parameters and **variable** for all others.
- If the method returns a value, it is translated either into a return value of the VHDL function or an additional output parameter of the VHDL procedure, depending on what kind of subprogram has been created. The above considerations on types, VHDL storage classes, and modes apply.
- With the exception of methods which access no attribute at all, the method's translated subprogram has an additional parameter. This parameter is used for passing the state of an object with which a method has been invoked to the method. Its properties are:
  - The name is THIS by default, but can be configured by the user in the translation tool that has been developed.
  - The type is an unconstrained std_logic_vector, making it possible to invoke the method's translated subprogram not only with an object of the class *C* in which the method is declared, but also with any derived class that has an extended state space, i.e., a state vector with a larger number of bits. This would not be possible if the parameter THIS were declared with the constrained std_logic_vector subtype that represents the encoded state of *C*.

- The mode is **in** if the method and all further methods invoked by it only read the object's attribute values. If the object's state is modified, the mode is **inout**. Note that the mode **out** is not used, even for methods that are write-only with respect to attributes, since out-mode parameter passing in VHDL involves an initialisation that would invalidate the value of unmodified attributes.
- The VHDL storage class is **constant** if the method is read-only with respect to the attributes, and **variable** otherwise.

For example, the following VHDL subprograms result from the translation of the methods that occur in the buffer example:

```
procedure put(
    variable THIS : inout std_logic_vector;
    constant val : in Integer );

procedure get(
    variable THIS : inout std_logic_vector;
    variable val : out Integer );

function is_full(
    constant THIS : in std_logic_vector )
    return Boolean;

function is_empty(
    constant THIS : in std_logic_vector )
    return Boolean;
```

7.3.2 Attribute access

The implementation of a method can read an attribute and assign it a value. A read access occurs when the attribute is used in an expression or as an actual input parameter in a method or subprogram call. The attribute's use as an assignment target or actual output parameter constitutes a write access. While parameter associations are dealt with in the following section, we here present the translation of attribute use in expressions and assignments.

VHDL code generation must consider that an attribute may be of a scalar or composite type, and that elements of composite types may in turn be composite. The translation of the respective access mechanisms is summarised in the following steps:

1) If the attribute is referenced by a simple name, this reference is replaced by a slice of the object's state vector from the attribute's start position $pos(i)$ to its end position $pos(i + 1) - 1$. Otherwise, it is replaced by a

subprogram call of Select_Element for read access or Assign_Element for write access, the before-mentioned slice is passed as an actual of the formal parameter A, and the following steps are executed:

2) If the attribute reference is an indexed name, i.e., a reference to an array element, the vectors passed as actual parameters in the access subprogram call are augmented as follows:
   - SIZE: The size $b(t)$ of the array element type is appended.
   - BASE: The lower bound *lb* of the array index range is appended.
   - INDEX: The index expression of the indexed name is appended.
   - OFFSET: A zero is appended.
3) If the attribute reference is a selected element, i.e., a reference to a record element, the argument vectors of the access subprogram call are updated as follows:
   - OFFSET: The element's start position $pos(i)$ is appended.
   - SIZE: The size $b(t_i)$ of the element's type is appended.
   - BASE, INDEX: A zero is appended to each.

The last two steps are repeated until no more indexed or selected element is found. Thereby, nested selection of record and array elements of complex data structures is translated. Other element access mechanisms, particularly in Objective VHDL, are pointer dereferentiation and slices that allow to access multiple array elements at a time. Both are considered as not synthesizable in VHDL [75] and therefore rejected by the object synthesizer.

As an example, we consider the first attribute of the class type *FIFO*. This attribute is an array with index range from 0 to *size*–1. To demonstrate nesting of composite types, we now assume that the array elements are records of type *REC* which in turn has three elements. Let us assume that the second element requires five bits. The write access to the second record element of the *i*-th array element of object *obj* is translated as follows:

```
Assign_Element(
    A => obj(obj'LEFT+FIFO_POS(1) to obj'LEFT+FIFO_POS(2)–1),
    INDEX => ( i, 0 ),
    SIZE => ( REC_SIZE, 5 ),
    BASE => ( 0, 0 ),
    OFFSET => ( 0, REC_POS(2) ),
    VALUE => <assigned_expression> );
```

The addition of obj'LEFT to the attribute position compensates the normalisation of the position values to FIFO_POS(1) = 0 (cf. section 7.2.3).

7.3.3 Intra-object method invocation

The term intra-object method invocation as coined in section 4.2.3 means that a method of an object invokes a method of the same object or of an exclusively owned sub-object. This kind of object communication, as it involves no parallelism and no need for request arbitration, can be translated into a VHDL subprogram call as follows:

- The called VHDL subprogram is the one that corresponds to the invoked method.
- The formal parameter THIS of the called subprogram is associated with the parameter THIS of the calling subprogram if a method of the same object is invoked. If a method is invoked with a sub-object, only that slice of the calling subprogram's THIS that corresponds to the attribute that holds the sub-object's state is passed to the called subprogram.
- Any association of an actual parameter with the method's formal parameters is transformed into an association of the actual with the corresponding formal of the translated VHDL subprogram.
- If a function method has been transformed into a VHDL procedure, we have to take into account that a procedure call cannot occur as part of an expression while the original function call can. This requires to invoke the procedure before the expression is evaluated, to assign the return value that is passed via an output parameter to a temporary variable, and to use this variable in the expression instead of the original function call. Note that in the presence of side effects in the function method, this code transformation may affect the expression result. Relying on a particular order of expression evaluation, however, is no good programming practice and the user should be prevented from doing so. This is ensured in Objective VHDL by not permitting side effects in function methods.

In all of the above-mentioned translated parameter associations, conversions or selection of an element of a composite type may be required. This is the case if, e.g., an indexed sub-object must be extracted from an array of sub-objects or if an element of a record-typed attribute is passed as an actual parameter to the invoked method. Two situations must be distinguished:

- If the formal parameter of the translated VHDL subprogram is an input parameter of VHDL storage class **constant**, an expression may be associated with it. This allows us to incorporate the required call of a conversion function or of the function Select_Element as part of the parameter association. Similarly, a return value of a function can be converted as required.

- If the formal parameter is an output or bidirectional parameter or of VHDL storage class **variable** (the latter condition coincides with the former ones in the translation presented), VHDL does not permit a conversion function call as part of the parameter association. A temporary variable of the formal parameter's type must be declared and associated with the formal. The variable is initialised with the original actual parameter just before the subprogram call. After the call, the variable's value is assigned to the original actual. Any conversions or selections required can be incorporated into these assignments.

For example, consider the following Objective VHDL method call, where in_p be an input parameter and bidir_p a bidirectional one:

```
subobject.method(
    in_p => some_array(i),
    bidir_p => some_record.element );
```

We assume that some_array and some_record have been encoded during translation, while the formal parameters in_p and bidir_p haven't been. Furthermore, be subobject the third attribute of the object. This leads to the generation of the following VHDL code:

```
variable temp : <type>; -- the type of bidir_p
...
temp := Stdv_to_<type>( Select_Element( some_record, ... ) );
method(
    THIS => this( this'LEFT + <>_POS(3) to this'LEFT + <>_POS(4)–1 ),
    in_p => Select_Element( some_array, i, ... ),
    bidir_p => temp );
Assign_Element(some_record, ..., <type>_to_Stdv(temp, <type>_size));
```

Analogous transformations, with the exception of not creating a parameter THIS, are required for normal subprogram calls whenever one of the actual parameters, after transformation into an encoded representation, must be converted into the type of the formal parameter.

7.3.4 Dynamic binding

A hardware implementation of dynamic binding of redefined methods has been devised in section 4.4.7. We now show how this structure (cf. figure 21) can be described by a VHDL subprogram so that its synthesis can be delegated to a behavioural synthesis tool:

- The dynamic binding subprogram is the same kind of VHDL subprogram (function or procedure) as the subprogram translation of the redefined methods.
- Its name is created by appending the suffix _DYNAMIC_BIND to the original method name. Further renaming is performed as required.
- The subprogram parameters of the dynamic binding subprogram, i.e. their names, types, modes, and VHDL storage classes, are the same as those of the translated method. Also the return type, if any, is the same.
- The subprogram body contains a **case** statement. The case expression selects the tag bits from the polymorphic object's state vector, which is passed as the parameter THIS. A slice is used to perform this selection, and must be qualified with the tag type (cf. section 7.2.3) to make the VHDL code legal.
- For each valid tag value, represented by a tag constant, a **when** branch is created. It invokes the subprogram translation of the redefined method version that corresponds to the tag. The following parameters are passed:
  - The formal parameter THIS of the invoked subprogram is associated with the object's state vector. If the called subprogram uses the self reference for dynamic binding, the complete state including the tag is passed to it. Otherwise, the state without the tag is selected by a slice of the parameter THIS of the dynamic binding subprogram.
  - The other parameters are associated with the corresponding parameters of the dynamic binding subprogram.
  - A return value (in case of a function) is passed on to the caller of the dynamic binding subprogram by a **return** statement.
- An assertion statement in the **others** branch reports any invalid tag values during simulation. This statement is ignored by synthesis.

The synthesis tool implements the functionality of every single branch, i.e. all the redefined versions of the method, in hardware. It creates as implementation of the case statement a multiplexer that selects the results of one of these sub-circuits according to the tag value. The fact that the execution of case branches is guaranteed to be mutually exclusive allows the synthesis back-end to implement resource sharing among the method sub-circuits.

The following listing presents these concepts with the dynamic binding subprogram that is generated for the method put of the buffer example:

```
procedure put_DYNAMIC_BIND(
    variable THIS : inout Std_logic_vector;
    constant val : in Integer ) is
begin
```

```
    case SYN_BUFFER_ALL_TAGS'( THIS(0 to 0) ) is
        when SYN_FIFO_TAG =>
            put_FIFO(
                THIS => THIS(1 to BUFFER_CLASS_SIZE),
                val => val );
        when SYN_LIFO_TAG =>
            put_LIFO(
                THIS => THIS(1 to BUFFER_CLASS_SIZE),
                val => val );
        when others =>
            assert false
                report "OVHDL runtime error: illegal or uninitialized tag"
                severity failure;
    end case;
end put_DYNAMIC_BIND;

-- procedure get_DYNAMIC_BIND : analogous
```

7.3.5 Example

As an example of the complete translation of a function and a procedure method, the VHDL code generated from the methods is_full and put of class LIFO is listed below. Their original code can be found in section 5.2.4.

```
function is_full(constant THIS : in Std_logic_vector) return Boolean is
begin
    return Stdv_to_Nat( THIS(THIS'LEFT + LIFO_POS(2) to
                             THIS'LEFT + LIFO_POS(3) – 1 ) = size;
end;

procedure put(
    variable THIS : inout Std_logic_vector;
    constant val : in Integer ) is
begin
    assert not is_full(THIS) report "LIFO overflow" severity failure;
    Assign_Element(
        THIS(  THIS'LEFT + LIFO_POS(1) to
               THIS'LEFT + LIFO_POS(2) – 1 ),
        Stdv_to_Nat( THIS(  THIS'LEFT + LIFO_POS(2) to
                            THIS'LEFT + LIFO_POS(3) – 1 ) ),
        bits,
        0,
        Int_to_Stdv( val, bits ) );
    THIS(THIS'LEFT + LIFO_POS(2) to THIS'LEFT + LIFO_POS(3) – 1)
```

```
        := Nat_to_Stdv(
                Stdv_to_Nat( THIS( THIS'LEFT + LIFO_POS(2) to
                    THIS'LEFT + LIFO_POS(3) – 1 ) + 1,
                intlog2( size + 1 );
end put;
```

7.4 Translation of objects

Given the bit-vector encoding of an object's state space and the translation of its methods into VHDL subprograms, we can now develop the generation of a synthesizable design entity. This entity-architecture pair is the translation of a class. Multiple architectures of the entity can be used to represent different optimizations. The component instantiation of such an entity-architecture pair corresponds to the (static) creation of an object in the OO model.

7.4.1 Interface of a synthesized object

The interface of a synthesized object is defined by a VHDL entity generated by the object synthesizer. This entity describes the inputs, outputs, and parameters of a circuit that implements an object of a non-derived or derived class as defined in section 4.4 (see figure 17 and figure 18, respectively):

- The entity name is the name of the class it implements. If it implements a class-wide type, the suffix _CLASS is appended. Further renaming is performed as required.
- All parameters of a class, including inherited ones, are represented as generics of the entity. This includes generic values of the OO model as well as generic types. The translation of a generic type passes the number of bits required to encode this type as an entity generic. A generic value can be implemented directly if it is an integer. Values of other types must be converted from their std_logic_vector representation into an integer value since only integer VHDL generics are synthesizable [75].
- Two input ports for global synchronous control signals are created; a clock signal, CLK, and a reset signal, RESET. Both are of type std_logic.
- The input ports SELECT_METHOD, IN_PARAMS, OUT_PARAMS, and DONE are defined to be connected to one ingoing channel. The DONE signal is of type std_logic. The other three signals are constrained std_logic_vectors whose width and encoding are as defined in section 4.4.3.

- For each outgoing channel (see section 5.3.5 for the determination of these channels from an Objective VHDL model), respective output signals (REQ, P_IN, P_OUT, DONE, cf. section 4.5.1) are defined.
- If all guard expressions are independent of the input parameters of a requested service, an RDY output port of std_logic_vector type is defined. The port has as many bits as there are valid service identifiers. The *i*-th bit is '1' if the guard expression of the service with identifier *i* is true, and '0' otherwise (cf. section 4.5.2).
- An auxiliary output port, STATE_OUT, may be defined to make the object's state observable outside of the object. This port is required if guard expressions depend on the services' input parameters and are therefore computed externally (cf. section 4.5.2).
- An auxiliary input port, STATE_IN, may be defined in order to improve the controllability of the object's state, e.g. for simulation purposes.

In the following, the interface of a polymorphic buffer object is listed. The class generics, size and bits, are translated into entity generics. There is one ingoing channel; arbitration of multiple clients is performed outside of the object if required. Outgoing channels are not required in this case as the object does not have external clients. The guard expressions are independent of input parameters; hence, an RDY port is created. The STATE_IN signal has been generated in this case in order to implement assignment (see section 7.4.3). The STATE_OUT output is only used for simulation purposes (observation of the state in a testbench).

```
entity BUFFER_CLASS is

    generic(  size : Positive;
              bits : Positive );

    port(  -- global control signals
           signal CLK   : in Std_logic;
           signal RESET : in Std_logic;
           -- ingoing channel
           signal SELECT_METHOD : in Std_logic_vector(0 to 1);
           signal IN_PARAMS : in Std_logic_vector( 0 to bits – 1 );
           signal OUT_PARAMS : out Std_logic_vector( 0 to bits – 1 );
           signal DONE  : out Std_logic;
           -- guard expression values
           signal RDY : out Std_logic_vector(0 to 3);
           -- auxiliary signals
           signal STATE_IN  : in Std_logic_vector(
                              0 to BUFFER_CLASS_SIZE(size, bits)–1 );
```

```
        signal STATE_OUT : out Std_logic_vector(
                                0 to BUFFER_CLASS_SIZE(size,bits)–1 ) );

    end BUFFER_CLASS;
```

7.4.2 Object architecture

An object entity may have several architectures corresponding to implementations of objects with different optimizations. An object architecture has a single process with code suitable for behavioural synthesis. The process contains declarations, initialisation code, and code that describes object functionality so that the structures presented in section 4.4 are synthesized from it. The process declarative part is logically split into the following parts:

- Declarations of constants that describe the encoding width and attribute positions (cf. section 7.2).
- The declaration of a type that represents the object's encoded state space.
- Declarations of VHDL subprograms that result from the translation of methods, including dynamic binding subprograms (cf. section 7.3).
- The declaration of a variable, named THIS, that stores the object state. Its type is the encoded class type. The variable is synthesized into the storage element of the object circuit.
- Declarations of auxiliary variables required for parameter passing.

The initialisation or reset behaviour is described by an outer infinite loop, labelled RESET_LOOP, as required by BC. For each attribute, the initial value defined in the OO model is assigned to the respective portion of the object's state vector. The techniques described in section 7.3.2 are used for implementing these assignments. Initialisation of array attributes may require the generation of a **for** loop. Furthermore, the entity's output ports are initialised. The DONE signal is set to '1', which means that the object is ready to accept requests. The guard expression values of the initial state are assigned to the RDY signal, if any, and the object's initial state value to the STATE_OUT signal, if any.

Regular object functionality is described by an inner infinite loop, labelled MAIN_LOOP, that is nested into the reset loop after the initialisation statements. Again, this matches the template required by BC. The object's operation includes waiting for a service request, executing the corresponding method, and updating the output signals. The code generated for that is as follows:

- A **case** statement queries the SELECT_METHOD input signal. It has a **when** branch for each method that can actually be called in the model and

an **others** branch. Methods that are never requested are not implemented. The multiplexer structure of an object circuit is synthesized from this statement.

- A **when** branch that corresponds to a method operates by setting the DONE signal low, invoking the VHDL subprogram that implements the requested service with the input parameter values (if any) received over the IN_PARAMS signal, and finally assigning the output parameter values (if any) to the OUT_PARAMS signal. Techniques for parameter association and type conversions are applied as described in section 7.3.3. If a method of a polymorphic object must be bound dynamically, its corresponding dynamic binding subprogram is called. From each branch, one method sub-circuit of an object circuit is synthesized.
- The **others** branch implements the object's idle mode by performing no operation. Since the process may run into this branch not only when no service is requested (SELECT_METHOD = 0, cf. section 4.4.3) but also if an invalid value is present on the SELECT_METHOD signal, we may add an assertion for detecting the latter situation. This is only for simulation; synthesis ignores the assertion statement.
- After the case statement, the object's state value is assigned to the STATE_OUT signal, if any, the guard expression values are converted from boolean to std_logic and assigned to the RDY output, if any, and the DONE signal is set high in order to indicate that the object is ready to accept the next request. After these operations, the end of the main loop is reached and its next iteration begins.

The resulting code structure is summarised in figure 43. A complete listing can be found in the next section.

In addition, statements of the form **wait until** clk'event **and** clk = '1' have to be generated. Each of them is followed by a statement that describes reset behaviour, **exit** RESET_LOOP **when** reset = '1', in order to let BC infer a global synchronous reset. These statements are required not only to satisfy tool requirements [152], but also to obtain a reasonable behaviour. Without any **wait** statement, VHDL simulation would execute the inner loop infinitely, never suspending the process and letting the simulated time proceed. In figure 43, these clock cycle boundaries are indicated by hatched bars.

- The first **wait** and **exit** statements must occur after initialisation and immediately before entering the main loop to express the synchronous start of normal operation.
- In a **when** branch that executes a method, the **wait** and **exit** statements must be inserted after the assignment to DONE and before reading the

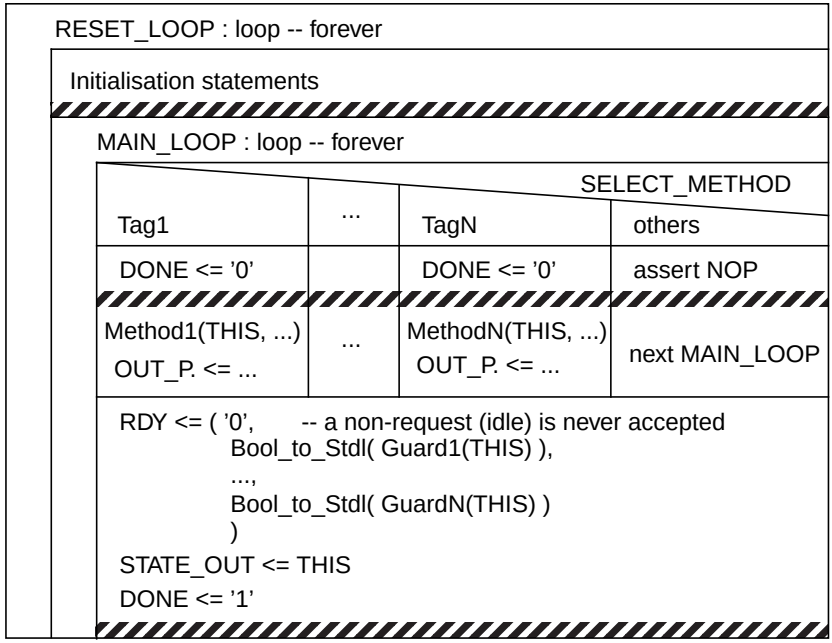


Fig. 43: Code structure of an object's process

IN_PARAMS signal. This satisfies the BC requirement that signal read operations be separated from preceding signal assignments by a clock cycle transition.

- BC demands that if at least one branch of an alternative (**if**, **case**) statement includes a wait statement, all others include one as well. Hence, a **wait** and **exit** must be generated in the **others** branch, too.
- Finally, a wait statement is generated before the end of the inner loop. This separates the assignments to output signals from read operations that occur when the next loop iteration starts, in this case the access to the SELECT_METHOD input in the case expression.

The final wait statement ensures that each loop iteration takes at least one clock cycle. Furthermore, the wait statements of the second and last point let consecutive assignments of '0' and '1' to the DONE signal take place in dif-

ferent control steps, but not in the same clock cycle. This ensures proper implementation of the handshake protocol defined in section 4.5.1.

However, the third kind of wait statement (wait in **others** branch) does not serve any purpose other than satisfying tool requirements on the VHDL code. It can even delay request acceptance unnecessarily. This happens when the **others** branch is entered because there is no request (SELECT_METHOD = “00“) in some cycle, and a request arrives in the next cycle. The object cannot respond in the cycle thereafter as it has to wait for two clock transitions, one in the **others** branch and the second at the end of the main loop, before the case statement is reached again.

This behaviour can be improved by adding a **next** MAIN_LOOP statement after the wait and exit statements of the **others** branch. Thereby, execution continues at the loop start, skipping the final wait and the preceding output assignments which are unnecessary as nothing has changed.

7.4.3 Example

The following listing shows the VHDL architecture generated as implementation of a polymorphic buffer object. The architecture contains a single process. In the declarative part of the process, declarations of constants, types, and subprograms are collected. These have already been presented as examples in section 7.2.4, section 7.3.1, section 7.3.4, and section 7.3.5. In addition, the object state variable THIS and a variable named OUTPUT_VAR which is to be used in the association of a subprogram output parameter (cf. section 7.3.3) are declared.

```
architecture BEHAVIOR of BUFFER_CLASS is
begin
    P : process
        -- declarations (constants, types, subprograms) mentioned before
        ...
        -- object state
        variable THIS : SYN_BUFFER_CLASS;
        -- auxiliary variables for parameter association
        variable OUTPUT_VAR : Integer;
```

The process’s statement part begins with the reset loop, where the object state is initialised as an empty FIFO buffer according to its initial value defined in the OO model. Likewise, all output signals are initialised. The initialisation part ends with a wait/exit statement combination.

```
begin
    RESET_LOOP : loop
        -- initialisation
        THIS(0 to 0) := FIFO_TAG;
        for i in 0 to size–1 loop  -- item := (others => 0)
            Assign_Element(
                THIS(1+FIFO_POS(1) to FIFO_POS(2)),
                (i, 0), (bits, bits), (0, 0), (0, 0),
                Nat_to_Stdv( 0, bits ) );
        end loop;
        THIS(1+FIFO_POS(2) to FIFO_POS(3)) :=     -- first := 0
            Nat_to_Stdv( 0, intlog2(size) );
        THIS(1+FIFO_POS(3) to FIFO_POS(4)) :=     -- nxt := 0
            Nat_to_Stdv( 0, intlog2(size) );
        THIS(1+FIFO_POS(4)) := Bool_to_Stdl(true); -- empty := true
        RDY <= ( '0', '0', '0', '1' );
        DONE <= '1';
        STATE_OUT <= THIS;
        wait until CLK'event and CLK = '1';
        exit RESET_LOOP when RESET = '1';
```

Next follows the main loop, whose first statement is a case statement that checks the SELECT_METHOD input. The following cases are distinguished:

- The method ID is “01“, which stands for put. The DONE signal is set low, and after a clock cycle transition the dynamic binding subprogram created for method put is invoked.
- The method ID is “10“, which stands for get. Operation is similar to the previous case. The auxiliary variable OUTPUT_VAR is associated to the subprogram’s output parameter. In an additional statement, its value is converted and assigned to the object entity’s parameter output signal, OUT_PARAMS.
- The method ID is “11“, which stands for assignment. After setting DONE low and awaiting a clock cycle transition, the value present at the object entity’s STATE_IN input is assigned to the object state variable, THIS. Note that without a dedicated STATE_IN input, the state value to be assigned would be transmitted over the IN_PARAMS input.
- The method ID has any other value. Error cases, e.g. the presence of a metalogical value such as 'X' (undefined), 'U' (uninitialised), or 'Z' (high impedance), are caught during simulation by an assertion statement. The only legal value is “00“, i.e. no request. If this value is present, the **next**

statement is reached after a clock wait, letting execution continue at the beginning of MAIN_LOOP.

Finally, the DONE signal is set to '1', the state output is updated with the object's state value, and the guard expression values are assigned to the RDY signal. After a clock wait, execution jumps back to the loop start.

```
MAIN_LOOP : loop
    -- normal operation
    case SELECT_METHOD is
        when "01" =>      -- implements dynamically bound put
            DONE <= '0';
            wait until CLK'event and CLK = '1';
            exit RESET_LOOP when RESET = '1';
            put_Dynamic_Bind(THIS, Stdv_to_Int(IN_PARAMS));
        when "10" =>      -- implements dynamically bound get
            DONE <= '0';
            wait until CLK'event and CLK = '1';
            exit RESET_LOOP when RESET = '1';
            get_Dynamic_Bind(THIS, OUTPUT_VAR);
            OUT_PARAMS <= Stdv_to_Int(OUTPUT_VAR, bits);
        when "11" =>          -- implements assignment
            DONE <= '0';
            wait until CLK'event and CLK = '1';
            exit RESET_LOOP when RESET = '1';
            THIS := STATE_IN;
        when others =>       -- implements idle mode
            assert SELECT_METHOD = "00"
                report "O-VHDL run time error: invalid method ID"
                severity failure;
            wait until CLK'event and CLK = '1';
            exit RESET_LOOP when RESET = '1';
            next MAIN_LOOP;
    end case;
    DONE <= '1';
    STATE_OUT <= THIS;
    RDY <= (  '0',                                  -- "guard" for idle
            Bool_to_Stdl( not is_full(THIS) ),      -- guard for put
            Bool_to_Stdl( not is_empty(THIS) ),     -- guard for get
                '1' );                              -- guard for :=
    wait until CLK'event and CLK = '1';
    exit RESET_LOOP when RESET = '1';
end loop MAIN_LOOP;
```

```
        end loop RESET_LOOP;
    end process ;
end BEHAVIOR ;
```

7.5 Inter-object communication

We have already explained in section 7.3.3 how method invocations within an object are translated into VHDL subprogram calls so that their synthesis is delegated to the VHDL synthesizer. Any inter-object communication between concurrent objects, however, must be dealt with by the object synthesizer since BC requires an explicit implementation of handshaking between concurrent processes. The most natural way of modelling the protocol involved would be its description as a finite state machine. Since this modelling style cannot be integrated with the sequential code that describes an object's behaviour, we must encode the FSM implicitly in the algorithm as shown in the following.

7.5.1 Remote method invocation by the client

In order to request a service from a server object, a client must execute the following steps (cf. section 4.5.1):

- Supply the encoded identifier of the service and the service's input parameters via the REQ and P_IN signals of the channel that addresses the server.
- Await the acceptance of its request, signalled by the channel's DONE signal going low.
- Reset the request immediately (i.e., in the next cycle) to no-operation in order to avoid it being accepted a second time.
- Await the completion of service execution, signalled by the channel's DONE signal going high.
- Read the output parameter values from the channel's P_OUT signal for later use. This must be done immediately as the P_OUT values are invalidated as soon as the server accepts the next request.

The second and fourth action, waiting for a specific value of the DONE signal, requires some further attention. We must take care for this action to be insensitive to glitches on the DONE signal, and for the generated code to be synthesizable. Following synchronous design principles, the first requirement is satisfied by observing the DONE value at the leading clock edge only. The second requirement forbids the use of a simple VHDL statement such as **wait**

on CLK **until** CLK = '1' **and** DONE = ... to express the desired functionality. Instead, busy waiting must be implemented using a loop. This and the other VHDL statements generated to implement the above steps are shown in figure 44.

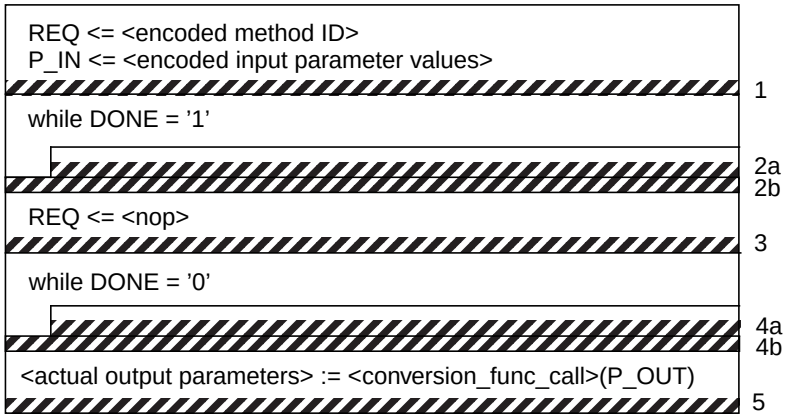


Fig. 44: Code inserted for requesting a service

The position of clock waits is indicated by hatched bars. The wait statements numbered 1 and 3 separate a write operation from a subsequent read operation (of the DONE signal in the while condition) as required by BC. Statements number 2a and 4a wait for one clock cycle inside their enclosing while loop. Thereby, the value of DONE is sampled after consecutive leading clock edges. Moreover, these wait statements ensure that time proceeds when simulating the busy waiting. Both while loops are immediately followed by further wait statements (number 2b and 4b). These result from the fact that BC schedules a loop exit as a control step of its own [152].

The code fragment depicted in figure 44 replaces a method call statement that is enclosed by other code. We must ensure its synthesizability and functional correctness in this context. To this end, the final wait statement (number 5) separates the assignment of actual output parameters from any possibly following read operation. The code start is not critical as the first operation, an assignment, may be preceded by read or write operations. For correct implementation of the handshake protocol, it is important that the DONE signal is high when entering the code. Otherwise, the REQ signal would be reset to the no-operation identifier too fast. The following properties ensure the desired functionality:

- The server object sets the DONE signal high during its initialisation.
- Any method invocation initiated by preceding code is finalised only after the DONE signal has gone high.
- Any preceding method invocation by another client is hidden by the arbiter, which keeps the DONE signal high on all channels to clients that are not being served.

Finally, we must consider that BC does not permit the access of signals such as DONE and REQ from a subprogram. Hence, if a remote method is invoked from a subprogram, e.g. from a VHDL subprogram that implements a method of the OO model, this subprogram must be inlined into the process from which it has been called. This means to replace a subprogram call by the sequential code of the subprogram body, to replace any access to a formal subprogram parameter by the actual parameter or expression, to include the subprogram's declarative part in the process declarative part, and to resolve any name conflicts by renaming. Furthermore, while allowing subprograms to be free of any clock waits, BC requires wait statements at certain locations in process code. Respective statements have to be added to the inlined code by the VHDL code generator.

7.5.2 Optimized protocol code

To implement a while loop of figure 44, the succeeding clock wait, and the following statements up to the next clock wait, VHDL code could be generated following the template listed below:

```
while DONE = <value> loop
    wait until CLK'event and CLK = '1';
    exit RESET_LOOP when RESET = '1';
end loop;
wait until CLK'event and CLK = '1';
exit RESET_LOOP when RESET = '1';
<subsequent statements>
wait until CLK'event and CLK = '1';
exit RESET_LOOP when RESET = '1';
```

This implies that any subsequent statements cannot be executed faster than one cycle after detection of the DONE value being waited for. For instance, REQ is reset to no-operation one cycle after DONE goes low. Since BC registers all outputs of a synthesized circuit, this response arrives at the server two cycles after service acceptance—too late according to our protocol definition

in section 4.5.1. To speed up timing, it would be desirable to execute subsequent statements in the same cycle loop termination is detected.

A straight-forward approach would be to move these statements between the **end loop** and the following clock wait and to remove the final clock wait of the code template. This is legal when using BC's superstate-fixed and free-floating scheduling modes, as opposed to cycle-fixed scheduling. However, BC always schedules a one cycle delay between loop termination and following actions.

The working solution is to move the subsequent statements so that they are executed before loop termination. This requires to describe loop termination explicitly using an **exit** statement in an infinite loop. The exit statement is preceded by the moved statements, and all these statements are conditional on the negated condition of the former while loop:

```
loop
    if DONE /= <value> then
        <subsequent statements>
        exit;
    end if;
    wait until CLK'event and CLK = '1';
    exit RESET_LOOP when RESET = '1';
end loop;
wait until CLK'event and CLK = '1';
exit RESET_LOOP when RESET = '1';
```

The above code template performs operations that logically follow the detection of a DONE value in the cycle the value is detected, which helps to speed up message exchange protocol sequences significantly. Further improvements are achieved in the case of successive remote method invocations by including the initial assignment statements (to REQ and P_IN, see figure 44) of the second protocol code template before the exit from the first one, which allows to save the first clock wait of the second template. Respective experiments will be made in section 8.4.3. Moreover, it may be possible to merge user-defined code that precedes or succeeds a remote method invocation with the generated VHDL code. This, however, depends on the surrounding statements.

7.5.3 Example

Let buffer_1 and buffer_2 be polymorphic objects of type Buffer_t'CLASS. The following Objective VHDL code describes the invocation of method get

with object buffer_1, where the actual output parameter is my_var_1. This is followed by the invocation of method put with object buffer_2, where the actual input parameter is my_var_2:

```
buffer_1.get( my_var_1 );
buffer_2.put( my_var_2 );
```

The following VHDL code implements this sequence by executing the remote method invocation protocol. The channel signals for addressing buffer1 and buffer2 are distinguished by appending respective suffixes, e.g. resulting in the names DONE_buffer_1 and DONE_buffer_2.

```
REQ_buffer_1 <= “10“                                    -- ID of method get
-- no input parameters
wait until CLK’event and CLK = ’1’;
exit RESET_LOOP when RESET = ’1’;
loop
    if DONE_buffer_1 /= ’1’ then
        REQ_buffer_1 <= “00“;
        exit;
    end if;
    wait until CLK’event and CLK = ’1’;
    exit RESET_LOOP when RESET = ’1’;
end loop;
wait until CLK’event and CLK = ’1’;
exit RESET_LOOP when RESET = ’1’;
loop
    if DONE_buffer1 /= ’0’ then
        my_var_1 <= Stdv_to_Int( P_OUT_buffer_1 );
        REQ_buffer_2 <= “01“;                           -- ID of method put
        P_IN_buffer_2 <= Int_to_Stdv( my_var_2, bits );
        exit;
    end if;
    wait until CLK’event and CLK = ’1’;
    exit RESET_LOOP when RESET = ’1’;
end loop;
wait until CLK’event and CLK = ’1’;
exit RESET_LOOP when RESET = ’1’;
loop
    if DONE_buffer_2 /= ’1’ then
        REQ_buffer_2 <= “00“;
        exit;
    end if;
```

```
        wait until CLK'event and CLK = '1';
        exit RESET_LOOP when RESET = '1';
    end loop;
    wait until CLK'event and CLK = '1';
    exit RESET_LOOP when RESET = '1';
    loop
        if DONE_buffer_2 /= '0' then
            -- no output parameter values to retrieve
            exit;
        end if;
        wait until CLK'event and CLK = '1';
        exit RESET_LOOP when RESET = '1';
    end loop;
    wait until CLK'event and CLK = '1';
    exit RESET_LOOP when RESET = '1';
```

The loop exits allow to implement a fast protocol as described in the previous section. Particularly, the finalisation of the first remote method invocation and the initiation of the second one are combined in the second loop. As both method invocations are perfectly independent of another, it is possible to even think about parallelizing them. Respective experiments are performed later in section 8.4.4.

7.5.4 Client-server interconnection

Having addressed VHDL code generation for a client's interaction with a server object, we shall now mention the interconnection of a client with its server. This means to generate a netlist that describes the instantiation and interconnection of object entities. VHDL netlists are dealt with in many textbooks, e.g. [9]. Hence, in this work, we only define the netlist's structure without going into the VHDL details:

- For each object $obj \in \Omega$, an instance of the entity-architecture pair that describes its optimized implementation is created.
- If the object has more than one client, i.e. there is more than one channel $(client, obj) \in \zeta$ in which *obj* is a server, an arbiter instance (cf. section 4.5.2) is created. The arbiter's generic number of client channels, *N*, is associated with the actual number of clients. The arbiter's server channel is connected to *obj*. The arbiter's scheduler component is configured according to a user-defined specification (e.g., using the VHDL attribute SCHEDULING of Objective VHDL as shown in section 5.4.2) or, in its absence, bound to a default scheduler circuit.

- For each channel $(client, server) \in \zeta$, the client object's corresponding output channel is connected either to an input channel of the server's arbiter or, in the absence of an arbiter, directly to the server's input channel.

Connecting an input channel and an output channel means the following:

- To instantiate REQ, DONE, P_IN and P_OUT signals and apply renaming as required.
- To connect these signals to the corresponding ports, REQ, DONE, P_IN, and P_OUT, that implement the output channel.
- If the input channel belongs to an arbiter, to connect these signals to a corresponding slice of the arbiter's C_REQ, C_DONE, C_P_IN, and C_P_OUT signals.
- If the input channel belongs to a server object, to associate REQ with the server's SELECT_METHOD signal, DONE to DONE, P_IN to IN_PARAMS, and P_OUT to OUT_PARAMS.

If the original OO model has a hierarchical structure beyond an object hierarchy, it is advisable to maintain this structure in the generated VHDL code. In Objective VHDL, a design hierarchy may be described by hierarchical instantiation of entities, where each entity may collect several objects that belong together. In this case, references to these objects (i.e., channels) are passed via entity ports. The generated VHDL code may maintain the original entity hierarchy. Channels or object references are implemented by replacing them by respective REQ, DONE, P_IN, and P_OUT port signals.

7.6 Summary

The generation of synthesizable, behavioural, algorithmic-level VHDL code from an object-oriented model has been presented following the structure of the meta-model of object-orientation presented in chapter 4. After developing a VHDL representation of the encoding of object data, the translation of methods has been addressed. The resulting declarations are collected and used in a VHDL design entity (entity-architecture pair) that represents a particular optimization of a class. Finally, we have described how the static structure of an object system, i.e. its objects and channels, is implemented by instantiating and interconnecting the generated design entities in VHDL.

Much effort has been devoted to the implementation of an optimized encoding of object-oriented data. Dealing with the encoding itself has been rather straight-forward, using induction over VHDL's type system of primitive and composite types plus Objective VHDL's object-oriented types. How-

ever, the necessary conversions from and into the original data types, particularly arrays, have required the application of sophisticated VHDL techniques such as synthesizable recursive subprograms.

Another important technical contribution is the development of the VHDL process that describes the object's behaviour and the integration of inter-object communication into this code. Finding code templates that describe efficient communication while being synthesizable and not triggering synthesis tool bugs has required a lot of experimentation. The presentation in this chapter highlights the results of this work, which indeed allow us to get from concurrent object-oriented models to a hardware implementation via the VHDL path as we will see in the next chapter.

In addition to the description of functionality in the form of an HDL model, constraints are an input to a synthesis tool. Currently, the designer must specify constraints on the generated VHDL code, e.g., in order to specify a maximum latency of the execution of a method or to define the resources available for object implementation. The automatic generation of HLS constraints from constraints on the object-oriented model is beyond the scope of this work, but may be a topic of future research.

Chapter 8

Experiments

While developing VHDL code generation, we have taken care of respecting the coding styles and templates required by the HLS tool we use for further synthesis. However, since these tool requirements are not formally defined (cf. the discussion in section 7.1.1) and since our VHDL code may trigger tool bugs, we have made some experiments to verify the practicability of our approach.

We must emphasize that a quantitative comparison of the quality of results or design time enabled by the object-oriented approach against a traditional design style is beyond the scope of our work. Rather, we have worked on a tool that is a prerequisite for such methodological considerations and perform these experiments to validate the techniques developed.

This chapter presents a selection of examples which systematically cover the different aspects of object-oriented models and their VHDL translation result. The experimental setup is explained in the first section. The remaining sections are devoted to the different experiments, namely the buffer example (continued), an extended buffer with sorting capabilities, and a class type that implements a synthesizable, binary fixed-point representation of real numbers.

Beyond validation of concepts, it is hoped that the simulation results presented in this chapter can deepen the understanding of aspects related to polymorphism, communication, arbitration and scheduling, and parallelism in object-oriented descriptions.

8.1 Setup

In this section, we detail the experimental setup by naming the tools and technologies used. Furthermore, we provide an overview of the experimental

coverage of concepts by listing all different aspects of the synthesis of object systems, respectively VHDL code generation, and correlating them with the sections of this chapter.

8.1.1 Tools and technology

Synthesis of the generated VHDL code has been performed using the Synopsys tool set, release 1998.08. Tasks performed in this environment include:

- Analysis and elaboration of the VHDL sources using VHDL compiler.
- High level synthesis including the steps constraint specification, scheduling, allocation, binding, and controller / datapath generation using Behavioral Compiler.
- Logic synthesis, optimization, and technology mapping with Design Compiler.

The design has been mapped to the technology library of the Altera FLEK10k FPGA. The area of synthesized circuits will be mentioned in units of the FPGA's logic elements (LEs). Each LE consists of a flip-flop and an SRAM based programmable four-input lookup table that can implement any boolean function of up to four variables. Since the purpose of our experiments is to validate the feasibility of synthesizing the generated VHDL code, the targeted technology is of minor significance within this work. It shall primarily enable a gate-level simulation of the synthesis results.

Pre- and post-synthesis simulations have been performed using the VHDL system simulator (VSS) from Synopsys.

8.1.2 Experimental coverage of concepts

We have developed VHDL code generation concepts under consideration of the VHDL RTL synthesis subset [75]. However, the VHDL subset accepted by behavioural synthesis is, while not formally defined, known not to be exactly the same as, nor a superset of, the RTL one. Furthermore, synthesis tool bugs may cause problems when following the proposed path to implement an object-oriented model in hardware. With these potential problems in mind, the experiments of this chapter are intended to validate the concepts developed and their practical feasibility.

The categories and concepts listed in figure 45 have been extracted from the sections of chapter 7; the exact place of their introduction is mentioned in a dedicated column. Another column lists the experiment, respectively the

section of this chapter, that covers the validation of the corresponding VHDL code generation concepts by synthesizing and simulating an example.

Category	Concept	Introduced	Experiment
Data types	scalar	7.2.1	8.2.1
	user-def'd enum	7.2.1	8.3.1
	composite	7.2.2	8.2.1
	class/derived	7.2.3	all
	polymorphic	7.2.3	8.2.2
Methods	interface	7.3.1	all
	attribute access	7.3.2	all
	local invocation	7.3.3	8.4
	remote invocation	7.5.1	all
	dynamic binding	7.3.4	8.2.2
Object	parameterization	7.4.1	8.2, 8.3
	interface	7.4.1	all
	body	7.4.2	all
	method selection	7.4.2	all
	inlining	7.5.1	8.4
Communication	interconnect	7.5.4	8.2
	protocol	7.5.1	all
	fast handshaking	7.5.2	8.4.3
	arbitration	4.5.2	8.2.4
	scheduling aspects	4.5.3	8.2.4
Optimization	relevant methods	7.4.2	8.4.2
	resource sharing	4.4.6	8.2.1, 8.3.3
	use of memories	4.4.4	8.2.3
	parallel requests	7.5.3	8.4.4
	setting constraints	7.6	all

Fig. 45: Concepts to be validated by synthesis experiments

8.2 The buffer example, continued

We continue the buffer example by synthesizing its components into gate netlists. A buffer object is simulated alone to demonstrate polymorphism and in combination with multiple concurrent clients to cover arbitration and scheduling. Moreover, we map the buffer's array for storing items to a memory so as to show that this HLS feature can be exploited in the context of object synthesis.

8.2.1 Synthesizing a polymorphic buffer object

Synthesis is performed in the following steps: It is assumed that the packages of the OO_SYN library (cf. appendix E) are already analysed. First, the files that contain the entity Buffer_Class and its architecture, BEHAVIOR, are analysed. Second, the design is elaborated for scheduling. This involves supplying actual values for all generic parameters so that VHDL elaboration can generate a non-parametric design instance. In the next step, design constraints are specified; in this example a clock period of 20 ns. After some checks and creating timing information for the design's operators, HLS is performed by the schedule command of Synopsys, using the superstate-fixed scheduling mode. The generated RTL implementation is finally compiled to gates and the resulting netlist written to a VHDL file for post-synthesis simulation. The synthesized design occupies 420 LEs. A dc_shell script that allows its reproduction is listed below:

```
analyze -format vhdl source/Buffer_Class.vhd
analyze -format vhdl source/Buffer_Class_BEHAVIOR.vhd
elaborate -s Buffer_Class -param "size = 8, bits = 3"
create_clock CLK -period 20
bc_check_design -io super
bc_time_design
schedule -io super
compile
write -format vhdl syn/Buffer_Class_SYN_BEHAVIOR.vhd
```

BC performs timing-driven scheduling by default, trying to reduce latency at the cost of more resources. In order to achieve resource sharing, we have to specify resource constraints and permit longer latency. This can be done by using Synopsys' set_common_resource command and the extend_latency option of the schedule command. Thereby, BC is forced into its resource-constrained mode. By allowing only a single resource for all addition and

subtraction operations, area is reduced to 336 LEs. In the implementation, the single resource is shared between the different methods (put and get) as well as between different method versions (for FIFO and LIFO) of the polymorphic object.

Likewise, schedulers and the arbiter circuit have been synthesized from RTL to gates for use in gate-level simulations. As an example of the synthesis results, a static priority scheduler and an arbiter with a maximum of four clients are displayed in figure 46. The scheduler requires 14 LEs and 12 LEs are needed for the arbiter. A round-robin scheduler has been synthesized into 35 LEs, and a modified round-robin policy that will be discussed in section 8.2.4 has been implemented in 52 LEs. These resource requirements can be considered as appropriately small compared to the circuits that implement the primary functionality.

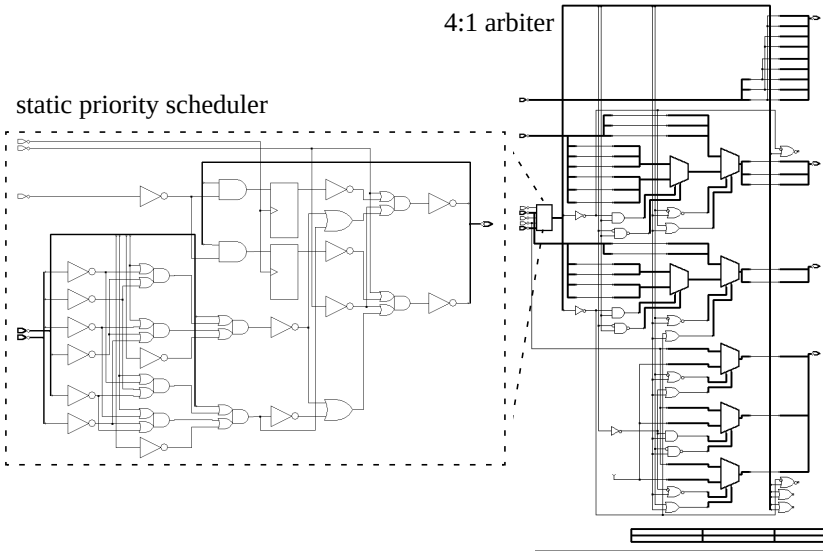


Fig. 46: Arbiter and scheduler circuit (4 clients)

8.2.2 Polymorphism

In order to demonstrate the effects and synthesis result of polymorphism, a polymorphic buffer with eight entries (parameter `size = 8`) is instantiated in a testbench as a server object. As there is only a single client, no arbitration is required. The testbench code executes the following scenario:

- The state value of an empty FIFO buffer is assigned to the object.
- A total of eight put requests is issued to the object. The values stored (parameter val of the method) are ascending from 0 to 7.
- A total of eight get requests is issued. The testbench verifies that the values are retrieved in first-in-first-out order, i.e. from 0 to 7.
- The object is assigned the state value of an empty LIFO buffer.
- Eight put requests with values ascending from 0 to 7 are issued.
- Finally, eight get requests are executed, asserting that values are returned in last-in-first-out order, i.e., from 7 down to 0.

Figure 47 shows waveforms from the gate level simulation of FIFO behaviour. The time unit is 1 ps. After an initialisation phase during which RESET is '1', the assignment method is requested from the server by the method ID 3 on the REQ signal. The server accepts the request and responds by setting the DONE signal low. The client, i.e. the testbench, acknowledges by resetting REQ to 0 (the no-operation ID). After completing service execution, the server sets DONE high.

This allows the client to issue its request of method put (ID 1 on the REQ signal) with value 0 (on the P_IN signal). Again, the handshake protocol is executed. When the service is finished at 250 ns, the new object state can be observed on the S_OUT signal, which is converted from the bit representation back to the original attributes' types in the testbench. Particularly, the attribute EMPTY transitions from true to false. In consequence, the RDY output, which represents the guard expression values, changes as well: RDY(2) goes high, signalling that from this point in simulation the method get (ID 2) can be invoked.

After seven more put requests, at 800 ns, the eight-item buffer is full. This is determined, according to method is_full, by the condition FIRST = NXT **and not** EMPTY, and signalled by RDY(1) going low. The testbench now continues with get requests. After the first get, the FIFO is no longer full and RDY(1) becomes high. After seven more get requests, at 1450 ns, the FIFO is empty so that no more get requests can be accepted and RDY(2) goes low. The values retrieved from the buffer are displayed on the P_OUT signal. Obviously, they are in first-in-first-out order as desired.

Further simulation results are displayed in figure 48: After a short pause, the testbench assigns a LIFO value to the polymorphic buffer object at 1550 ns; see the REQ and S_IN signals. After the completion of this service, at 1600 ns, the buffer operates in its last-in-first-out mode. A vertical line indicates the boundary between FIFO and LIFO behaviour.

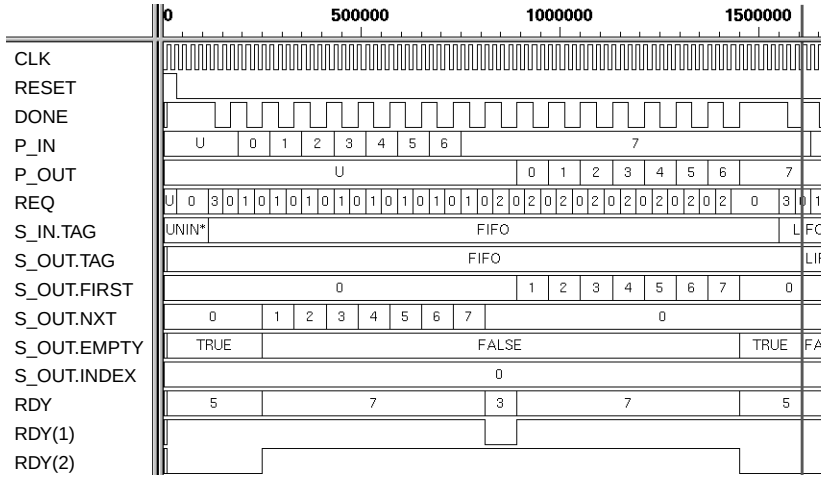


Fig. 47: Polymorphic buffer operating as FIFO

The remaining simulation is similar to what has been explained before: eight put requests, letting the buffer run full, followed by eight get requests, letting the buffer run empty. However, as expected, the stored values are now being output in reverse, last-in-first-out order.

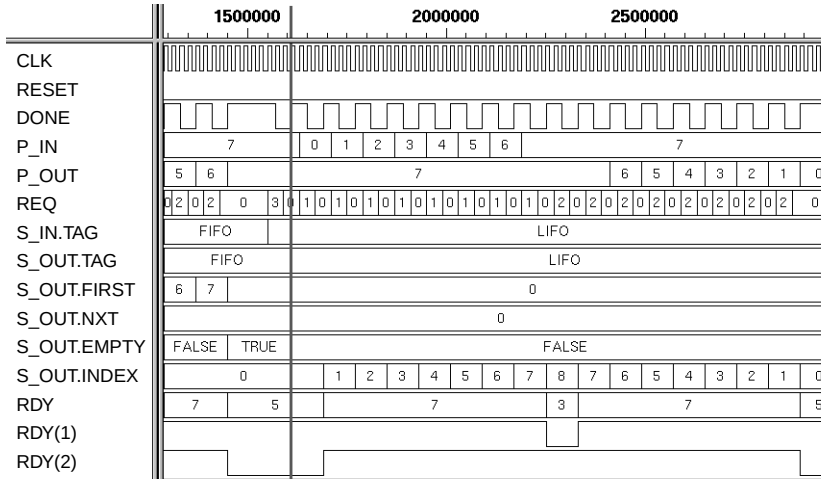


Fig. 48: Polymorphic buffer operating as LIFO

8.2.3 Mapping attributes to memory

Instead of storing all object data in registers, it may be desirable to map one or several of an object's attributes to a memory component, which may be a register file, SRAM, or DRAM. This approach has the following advantages:

- A dedicated memory block is typically more efficient in terms of area, timing, and power than a synthesized set of registers.
- Inference of pre-defined memories helps avoid degradation of the synthesis tool's performance when large arrays must be implemented.

On the other hand, the use of memories reduces the potential for concurrency compared to a set of individual registers. This must be considered by the user when selecting a memory type, particularly to provide an appropriate number of memory access ports. Furthermore, the concurrency aspect influences the choice of attributes to be stored in memory.

In the buffer example, it is reasonable to map the attribute item to memory for the following reasons:

- This choice avoids logic and synthesis performance degradation when the parameters size or bits are large.
- The homogeneous data type of item, an array of elements which are of the same size, suits the memory structure well.
- The methods contain only sequential accesses of item; no potential for concurrency is lost.

To use the memory inference feature of BC with an attribute, slightly modified VHDL code must be generated. We must exclude the attribute from the variable THIS and its std_logic_vector subtype (cf. section 7.2.3 and 7.4.2). Instead, the attribute is to be implemented as an individual variable of an array type. It is possible not to encode this type at the bit level, but to maintain its original, high-level data type. In addition, two attributes and a dedicated memory resource must be declared in the specific way required by BC to recognise memory variables. For the buffer's attribute item, the following declarations must be added in the declarative part of the object's process (cf. section 7.4.2):

```
constant RAM : resource := 0;
attribute variables of RAM : constant is "item";
attribute map_to_module of RAM : constant is "DW_ram_r_w_s_dff";
variable item : Buffer_array;
```

The constant RAM tells BC to instantiate one resource; its value is irrelevant. The VHDL attribute variables specifies the variables to be mapped to the RAM resource. The VHDL attribute map_to_module specifies the component

that implements the resource; in this case an SRAM with one read and one write port from the Synopsys Designware library. Finally, the attribute item is translated into a variable of its original type (see section 7.2.4 for the declaration of Buffer_array). In addition, library and use clauses must be included to make Designware and Synopsys' behavioural and attribute declarations visible:

```
library SYNOPSYS; use SYNOPSYS.ATTRIBUTES.all;
                  use SYNOPSYS.BEHAVIORAL.all;

library DWARE;    use DWARE.DWPACKAGES.all;

library DW06;     -- contains the particular memory used
  use DW06.DW06_COMPONENTS.all;
```

In the subprograms that implement the methods, all accesses to the memory-mapped attributes are left in their original form instead of being translated into accesses of a slice of the parameter THIS. Instead of the attribute item of the object-oriented buffer model, the variable item of the object process is now accessed. Note that BC does not permit to pass a memory variable as a parameter; it must be accessed directly by the method subprograms. In VHDL, this is allowed only for procedures. Hence, any function method that uses the variable would have to be translated into a procedure as described in section 7.3.1.

From this modified VHDL code, the polymorphic buffer can be synthesized using the map-to-memory feature of BC. The circuit requires 217 LEs, including the memory, which is about half the size compared to direct synthesis of the array. Simulation confirms the correct functionality of the resulting netlist. Even the timing at the object's interface ports is, in this case, the same. In general, however, a sequentialization of memory accesses could cause methods to require more clock cycles for execution. Still, the inter-object handshake protocol would ensure the correct communication of results to the client.

8.2.4 Arbitration

We shall now demonstrate the arbitration of concurrent requests. To this end, another testbench of the synthesized buffer has been designed. This testbench instantiates an empty FIFO server object. There are three clients (concurrent processes of the testbench) which, at the beginning, have either no request or an unacceptable put request. Then, starting at 200 ns, the following happens:

- The first client, C1, requests the assignment method (implicitly defined in Objective VHDL). After completion of this service, C1 sends requests all the time to put the value 1 into the buffer.
- The second client, C2, continuously issues put requests with value 2.
- The third client, C3, requests the get service all the time.

The behaviour of the arbitrated buffer is illustrated by tracing the following signals:

- The request sent by each client to the arbiter (C1_REQ, C2_REQ, C3_REQ) and the respective done signal returned by the arbiter (C1_DONE, C2_DONE, C3_DONE).
- The RDY signal over which the arbiter receives guard expression values from the server.
- The internal GRANT signal of the arbiter, carrying the number of the client whose request is granted, or the value 0 if none.
- Finally, the request, data input, data output, and done signals by which the arbiter communicates with the server: S_REQ, S_P_IN, S_P_OUT, and S_DONE.

The first simulation has been configured to apply static priority scheduling where priority increases with the client's number (C3 has higher priority than C2; C2 has higher priority than C1). This results in the waveform of figure 49. As we would expect, no request is granted as long as there are only

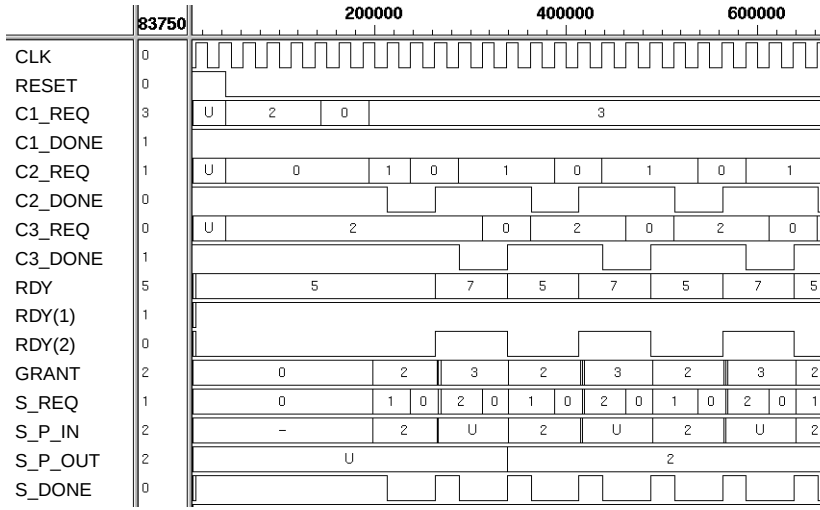


Fig. 49: Static priority scheduling

unacceptable or idle requests. At 200 ns, both C1 and C2 issue a request that can be accepted while C3's get request still cannot be serviced as the buffer is empty. Among the clients with acceptable requests, C2 has the highest priority; hence, its request is granted (see the value 2 on the GRANT signal). The handshake protocol is executed on the signals C2_REQ and C2_DONE, which are connected to S_REQ and S_DONE by the arbiter, and the value 2 (see S_D_IN) is stored in the buffer.

This makes C3's get request acceptable. As C3 has highest priority, its request is indeed executed at 300 ns, and the value that has just been stored is retrieved back from the buffer (see the value 2 on S_P_OUT). Afterwards, the buffer is empty and C2 again becomes the highest-priority client that has an acceptable request. Hence, C2 and C3 continue to be served alternately forever. Obviously, this is not a fair schedule as C1 suffers starvation.

In an attempt to establish fair scheduling, we now try a scheduler that follows a round-robin policy, attempting to serve the clients' requests in a circular order. If it is a particular client's turn, but its request cannot be accepted at that time, this client is skipped. The resulting system behaviour is displayed in figure 50. As long as the buffer is not full, the new scheduling strategy works well, serving one client after the other. Since only one get service but two put services are executed each round, the buffer runs full at 1900 ns. This time is indicated by the vertical line.

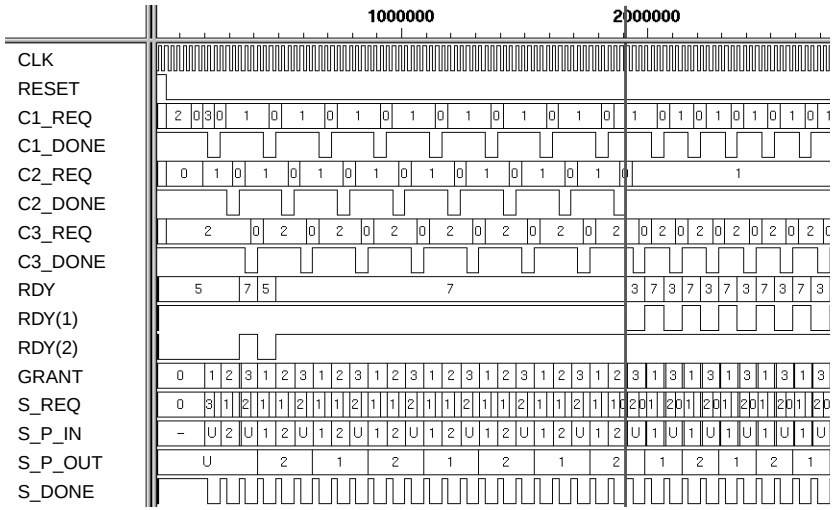


Fig. 50: Round robin scheduling

Later, only a single put request can be accepted after each get. According to the round robin scheme, this is always the request from C1 since C1 follows C3. When it is C2's turn, the buffer is full and C2 must be skipped. Hence, C2 suffers starvation from time 1900 ns on.

The problem in this case is that after accepting a request, the round robin policy continues with the client that follows the one that has been serviced. A modified scheduler could keep track of the first unsuccessful client, i.e. one that had to be skipped for an unacceptable request, and resume from this one in order to avoid its starvation. Such a scheduler has been implemented and synthesized. The simulation result is displayed in figure 51. It equals figure 50 until the buffer runs full at 1900 ns. Then, the following happens:

- C3's get request is accepted.
- C1's put request is accepted.
- C2's put request cannot be accepted. The scheduler skips C2 but stores it to be considered first the next time.
- C3's put request is accepted.
- The next round starts from C2, whose request can be accepted now.
- This sequence starts again from the first point and repeats infinitely.

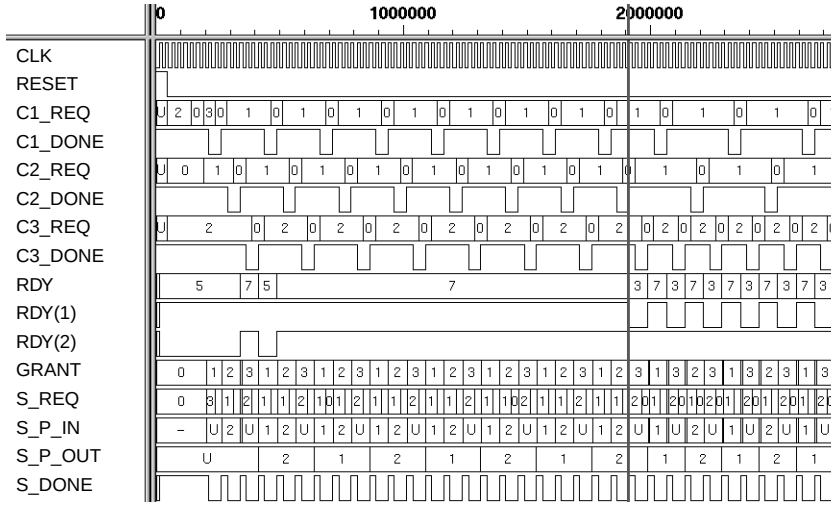


Fig. 51: Scheduling for fair arbitration

Since all clients are serviced in a finite, repeated sequence (C3-C1-C3-C2), no client is blocked indefinitely. The scheduler avoids starvation in this case. Still, a different sequence could be desirable depending on the application.

Hence, the choice of a scheduling policy, and possibly the implementation of a scheduler circuit if not yet available, must be left to the user.

8.3 A sorted buffer

In this section, the put method of a buffer is implemented so that it sorts the buffer entries. This allows us to present the synthesis and simulation of a complex method with data-dependent latency.

8.3.1 Modelling in Objective VHDL

The sorted buffer is, like FIFO and LIFO, another class type derived from the class Buffer_t. It inherits from its parent class the attribute item and the method declarations is_full, is_empty, put, and get. The methods have no implementation yet. Hence, they are implemented in the body of class Sorted_Buffer. Put implements a sorting-by-insertion algorithm. It inserts its argument, val, into the item array so that this array remains sorted, i.e., an element at a lower index is smaller than or equal to an element at a higher index. Get retrieves either the largest (top) element or the smallest (bottom) element from the buffer.

The specific mode of operation of get can be defined by invoking the method set_mode (see the listing below) with a parameter value of the user-defined enumeration type Mode_t, either get_top or get_bottom. The corresponding functionality is implemented by the new methods get_bottom and get_top. Note that these names are overloaded with the respective homographs defined by the enumeration literals.

The class Sorted_Buffer has attributes named bottom_index, top_index, and empty, which are similar to the FIFO's first, nxt, and empty. Another attribute, mode, stores the mode of operation set by set_mode.

```
type Mode_t is ( get_top, get_bottom );

type Sorted_Buffer is new class Buffer_t with

    class attribute bottom_index   : Integer range 0 to size – 1 := 0;
    class attribute top_index      : Integer range 0 to size – 1 := 0;
    class attribute empty          : Boolean := true;
    class attribute mode           : Mode_t := get_top;

    function is_full return Boolean;
    function is_empty return Boolean;
```

```
    for variable
        procedure put( val : Integer range 0 to 2**bits – 1 );
        procedure get( val : out Integer range 0 to 2**bits – 1 );
        procedure get_bottom( val : out Integer range 0 to 2**bits – 1 );
        procedure get_top( val : out Integer range 0 to 2**bits – 1 );
        procedure set_mode( m : Mode_t );
    end for;

end class Sorted_Buffer;
```

For space constraints, we do not go into all implementation details of the class body. Only some algorithmic aspects shall be touched on. The put method contains the following while loop to find the position at which the new value, val, is to be inserted:

```
index := bottom_index;
while index mod size /= top_index and item(index) < val loop
    index := (index + 1) mod size;
end loop;
```

Another while loop shifts all above items for one position in order to make room for the value to be inserted:

```
index2 := top_index;
while index2 /= index loop
    item( (index2 + 1) mod size ) := item(index2);
    index2 := (index2 – 1) mod size;
end loop;
```

Clearly, these loops are data-dependent. Their number of iterations varies with the value to be inserted and depends on the object's state, i.e. the number and values of items already in the buffer. We consider some implications later.

8.3.2 VHDL code generation

VHDL code that describes a synthesizable implementation of the sorted buffer can be generated according to chapter 7. However, we must pay attention to the subprogram that implements the put method. Since this subprogram contains the above loops, it would be rejected by BC. A workaround is to inline the subprogram into the object process code as described in section 7.5.1.

Inlining can be avoided when the translated subprogram is marked as a scheduled subprogram using a tool-specific pragma (meta-comment):

```
procedure put(   variable THIS : inout SYN_Sorted_Buffer;
                 constant val : in Integer range 0 to 2**bits – 1 ) is
-- synopsys preserve_schedule_subprogram
...
```

This subprogram is scheduled independent of the process from which it is called and is treated as a black box thereafter. While this limits the optimization potential compared to a global schedule, the complexity of scheduling is reduced.

However, the BC documentation is not clear about whether or not an inout parameter such as THIS is permitted with a scheduled subprogram. The description mentions no such restriction, but lists an error message (number HLS-2) for this case. In version 1998.08, BC accepts the above subprogram, but cautious validation of the synthesis result seems advisable.

8.3.3 Synthesis

A sorted buffer with eight entries has been synthesized similar to the FIFO / LIFO buffers. The implementation generated by timing-driven scheduling takes 445 LEs. Resource-driven scheduling with constraints allowing only a single adder/subtractor can be performed, but results in a larger design of 460 LEs due to multiplexer and control overhead.

However, we did not succeed in generating a correct implementation with the item array mapped to memory. The cause appears to be a bug in the synthesis tool; an examination of the simulation of the generated netlist reveals that some array accesses are scheduled out-of-order when memory is used. An error in the VHDL input description can be ruled out since it synthesizes correctly when the array is not mapped to memory.

8.3.4 Simulation

The synthesized sorted buffer has been simulated in a testbench that executes the following scenario:

- Put requests with numbers in the permitted range are sent to the object until the buffer is full. In the case presented here, the values are 0, 3, 6, 2, 5, 1, 4, and 0 (in this order).
- The values are retrieved back by get requests. Since the default mode is get_top, we expect to receive the values in highest-to-lowest order.
- The above values are put into the buffer again.
- The buffer's mode is set to get_bottom by requesting set_mode.

- Get requests are issued. The values should be received in lowest-to-highest order.

A waveform of the simulation of the first two points is displayed in figure 52. A ’1’ on the REQ signal stands for a put request, and a ’2’ represents a get request. The put requests take place from time 0 up to the vertical bar shortly after 14,000 ns. Later, eight get requests are issued and indeed the values are returned as expected. Note that at the right hand of the waveform, two successive zeroes are returned. While there is no change on the P_OUT signal, these values can be differentiated by observing the P_OUT values shortly after each rising edge of the DONE signal.

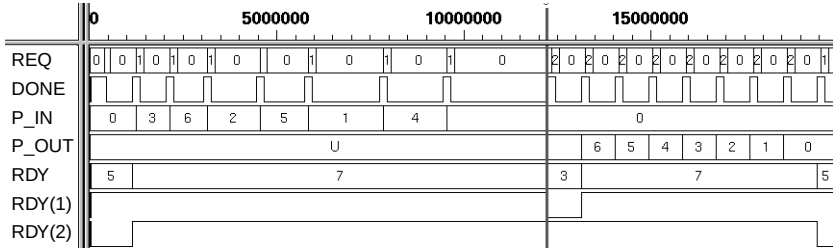


Fig. 52: Simulation of sorted buffer

The remaining simulation is shown in figure 53. We see the same put requests again, followed by a set_mode request that is identified by the value 3 on the REQ signal. Finally, there are another eight get requests, and the values 0, 0, 1, 2, 3, 4, 5, 6 appear on the P_OUT signal as expected. Again, we must look at the DONE signal to identify the point in time when a value is valid. The first valid value is marked by the vertical line.

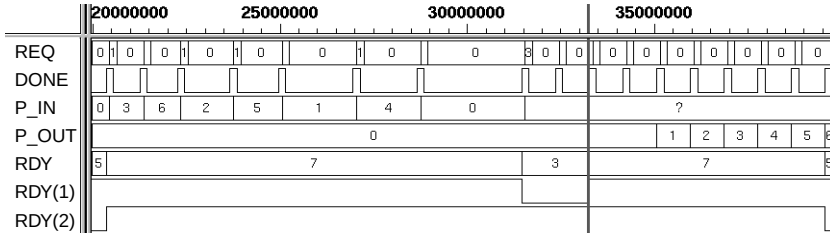


Fig. 53: Simulation of sorted buffer (cont’d)

This simulation confirms that and shows how the handshake protocol on the REQ and DONE signals synchronises the client with the execution of a service (here: put) that has a data-dependent latency.

8.4 Fixed point arithmetics

This section presents an excerpt of a data type for fixed point arithmetics and an example of its use. The example allows us to give emphasis to the difference between exclusively owned subobjects and external server objects. Moreover, we will address communication protocol and parallelism issues.

8.4.1 A fixed point class type and its use

The class type Fixed_Real listed below describes a binary fixed-point representation of numbers. It allows to define by respective generics the number of bits used to represent an integer and a fractional part. A method set is provided to initialise a Fixed_Real object. Further methods, add and mult, implement two basic arithmetic operations.

```
type Fixed_Real is class

    generic( integer_bits, fraction_bits : Natural );
    ... (class attributes and functions omitted for brevity)
    for variable
        procedure set( neg : Boolean; int, frac : Natural );
        procedure add( val : Fixed_Real );
        procedure mult( val : Fixed_Real );
    end for;

end class;
```

To demonstrate the use of such a type, a class that computes values of a geometric series has been implemented. This application has been chosen because it allows to emphasize with a brief piece of code all the aspects to be addressed: subobjects vs. concurrent objects, faster inter-object communication, parallel service execution, and pipelining.

The Geometric_Series class has a single method, compute, with parameters start and factor of type Fixed_Real_4_4 which is defined as Fixed_Real with four integer bits and four fraction bits. This method computes the series defined by

$$sum_n = \sum_{i=0}^{n} start \cdot factor^i . \tag{89}$$

This is done in an iterative way, keeping the value of $start \cdot factor^i$ in a variable named product, and the successive sum values in a variable named

sum. Iteration is stopped when sum changes no longer, which happens due to limited accuracy, or by an overflow exception in the case of divergence. Sum and product may either be subobjects of class Geometric_Series or external objects. In the first case, they are implemented as class attributes (cf. section 4.1.7). In the latter case, they are made known to Geometric_Series by passing them to its method as signal parameters (cf. section 5.3.5). Both alternatives are mentioned by means of comments in the following listing:

```
type Fixed_Real_4_4 is Fixed_Real generic map( 4, 4 );

type Geometric_Series is class

    class attribute sum_old : Fixed_Real_4_4;
    -- the following attributes only when product and sum are sub-objects
    -- class attribute product : Fixed_Real_4_4;
    -- class attribute sum : Fixed_Real_4_4;

    for variable
        procedure compute(
            constant start, factor : Fixed_Real_4_4;
            -- the following parameters only when product and sum
            -- are external objects
            signal product, sum  : inout Fixed_Real_4_4 );
    end for;

end class;
```

The code shown below implements the method compute. Two aspects will be important in later considerations:

- The first two statements are independent of another and could be executed in parallel.
- Inside the while loop, the add operation is data-dependent on the result of the preceding mult operation. Hence, the addition must be executed after the multiplication. It can, however, be executed in parallel with the multiplication of the next loop iteration. This is known as loop pipelining.

```
product.add(start);
sum.add(start);
WHILE_CHANGE : loop
      product.mult(factor);
      sum.add(product);
      exit WHILE_CHANGE when sum_old = sum;
      sum_old := sum;
end loop;
```

8.4.2 Synthesis

Three different versions of VHDL code implementing the Geometric_Series object have been generated and synthesized:

- A Geometric_Series architecture with non-optimized handshaking as in section 7.5.2 (172 LEs), connected to external sum and product objects that implement all methods (508 LEs each). This results in a total of 1188 LEs.
- A Geometric_Series architecture with fast handshaking as in section 7.5.1, (154 LEs) and external sum and product objects that implement only the methods that are actually invoked (sum: add method, 160 LEs; product: add and mult methods, 432 LEs). The complete design has 746 LEs.
- A Geometric_Series architecture with exclusively owned subobjects, synthesized into 753 LEs.

The second version has a clear area advantage over the first one, which emphasizes the importance of implementing only the methods that are actually invoked with each individual object. In the third case, the purely sequential VHDL code has enabled BC to determine and leave out methods that are never called, leading to a similar area result.

Obviously, the fast handshaking has a slight area advantage over non-optimized message exchange. This has its roots in the smaller number of control states required to implement the optimized protocol.

8.4.3 Simulation

The different versions of a synthesized Geometric_Series object have been simulated, invoking method compute with the parameters start = 4.0 and factor = 0.75. The development of the sum and product values can be observed on the S_SUM and S_PRODUCT signals of figure 54. Note that the fractional part is displayed in units of $1/16$. Due to limited accuracy, iteration stops at sum = 15.3125, whereas the infinite series' value is 16.

Figure 54 shows the non-optimized handshaking between the Geometric_Series object and its external servers, product and sum. We can observe that services are requested from these servers alternately and in a non-overlapping way although, as explained in section 8.4.1, they could potentially be executed in parallel.

The simulation of fast handshaking is shown in figure 55. We see that the optimized protocol implementation reduces latency significantly; the compute method is finished after 8,575 ns instead of 14,225 ns, although the

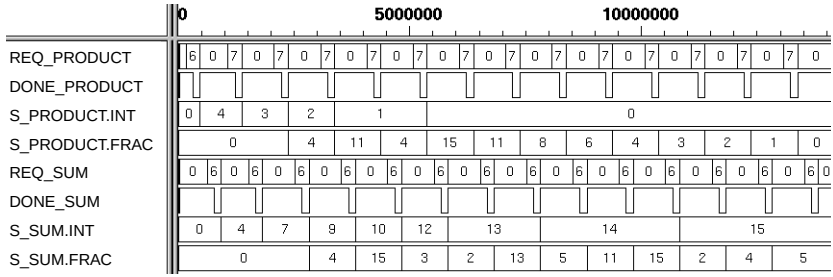


Fig. 54: Geometric series with slow handshaking

clock cycle time is the same (50 ns) in both cases. This has its roots in the faster response of the communication partners. We can see this fact from the servers' idle time, the period during which their DONE signal is high, which is reduced from 850 ns (figure 54) to 450 ns (figure 55)[1]. Still, the potential for concurrency is not exploited since only one server is busy at a time.

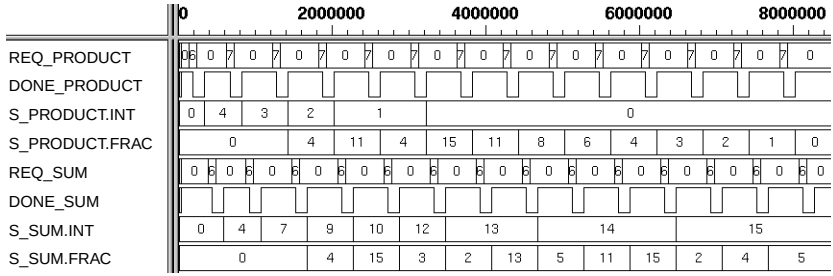


Fig. 55: Geometric series with fast handshaking

The simulation of the Geometric_Series object with exclusively owned subobjects is not shown in a figure because there is no handshaking; all functionality is implemented by BC in a single datapath and controller. The results of computation are the same as before while latency is reduced to 2,675 ns for two reasons. First, there is no more handshaking overhead. Secondly, BC can analyse the purely sequential algorithm at the level of its schedulable operations and at least partially parallelize the method invocations of the two subobjects.

1. Note that the time scales of these figures are different.

8.4.4 Parallelization attempts

While we cannot avoid the handshaking overhead when requesting services with unknown latency from external, concurrent servers, we can try to achieve more parallelism of remote services. It is important to note that the sending of these requests is defined by the *sequential* algorithm of the client object. However, if the requests address different concurrent servers, they can potentially be parallelized. A precondition for such parallelization is the absence of data dependencies between the services. Furthermore, it should be ensured that the introduction of concurrency does not introduce deadlocks into the system. While respective analyses are beyond the scope of this work, we have evaluated the possibility of using BC for synthesizing parallel inter-object communication from the generated VHDL code.

The first experiment has been performed with the first two statements of the method compute (see section 8.4.1). VHDL code generation (see section 7.5.1) produces for each of these service requests two loops that implement handshaking. These loops are symbolised as L1, ..., L4 in the following:

```
product.add(start);   —translation—>  L1; L2;
sum.add(start);       —translation—>  L3; L4;
```

Parallelizing the first and the second request means to execute L1 and L2 in parallel to L3 and L4 while maintaining the sequences L2 after L1 and L4 after L3:

```
L1; L2; L3; L4;       —parallelisation—>  (L1; L2) || (L3; L4)
```

However, BC's superstate-fixed scheduling mode, which has been used up to now, maintains the sequence of all IO operations such as the handshaking code found inside the loops. A reordering can only be performed in the free-floating scheduling mode. Hence, this mode has been applied. Since it completely ignores any control dependencies, observing data-dependencies only, additional precedence constraints had to be defined so as to maintain the sequence of operations that must not be parallelized. In particular, such constraints have been set between L1 and L2 as well as between L3 and L4 by means of BC's set_min_cycles command. On the other hand, no such constraints have been set between L1 and L3, L1 and L4, L2 and L3, and L2 and L4.

With these settings, BC did still schedule the four loops one after the other. The same occurs even with an additional constraint using BC's set_cycles command to force the start of the loops L1 and L3 into the same

cycle. It appears that BC does not schedule loops to be executed in parallel, but the tool documentation does neither confirm nor deny this.

The idea of pipelining the algorithm's WHILE_CHANGE loop (cf. section 8.4.1) had to be discarded because BC only pipelines loops that contain no other loops. During translation, however, the above loops L1, ..., L4 are inserted into WHILE_CHANGE to implement communication.

8.5 Summary

Our experimental results confirm that the path from object-oriented models to a hardware implementation suggested in this work is practicable. The mechanisms suggested to implement objects, polymorphism, communication, and arbitration are all functional after synthesis to the gate level.

More evaluations of Objective VHDL and its translation into VHDL, mostly with simulation emphasis, can be found in the following publications: In [5], the modelling of ATM cell data is presented. A portal crane controller has been designed as a contribution to [113]. The REQUEST project's [111] evaluation reports on a basic reuse library and its utilisation in the domain of digital filters [136], pointing out the importance of generic modelling, which is fully supported by this synthesis approach.

The importance of request scheduling has been highlighted with the buffer example. When modelling concurrent objects, the user must be aware of the need to provide guidance to enable an appropriate scheduling. The integration with our synthesis approach is ensured by the specification of a scheduling strategy in the object-oriented model and its mapping to an externally provided component (cf. section 5.4.2).

As we have seen, the synthesis of sequential algorithms can be delegated to a HLS tool. This allows to employ features such as resource sharing and the mapping of data to a memory instead of a register bank. However, it has not been possible to utilise HLS for parallelizing or pipelining communications with concurrent server objects. At the moment, it is advisable to keep the object-oriented synthesis model as sequential as possible, using subobjects wherever appropriate. Future research should advance the analysis and code generation techniques of this work towards an optimized communication implementation. This could involve considering techniques such as relative scheduling [85] and multiple-process synthesis [47] for implementing remote method invocation.

Chapter 9

Conclusions

This thesis describes the generation of optimized digital circuit structures from object-oriented synthesis models.

Object-orientation can be utilised to design software and hardware at a high level of abstraction and with support for reuse and extension of functionality. The use of object-oriented programming languages for software implementation and their automatic compilation into executable code are state-of-the-art. Our work targets at developing respective techniques for implementing the hardware part of a system.

While there exist a variety of research activities on object-oriented hardware description and synthesis, using both object-oriented hardware description and programming languages, the following aspects are the distinguishing features of this work:

- A language-independent formulation based on a meta-model of object-orientation.
- Wide synthesis support for all major object-oriented features: objects, classification, genericity, inheritance, polymorphism, communication by message passing, and dynamic binding.
- Explicit description of the synthesized circuit structures.
- Optimization of value-based polymorphism.
- Synthesis of communication between concurrent objects, including arbitration of concurrent requests and condition synchronisation, from a description that avoids issues in the interplay of inheritance and concurrency with the help of the guarded method concept.

The meta-model of object-orientation has been designed to cover the static modelling features of the Unified Modeling Language, a de facto standard diagrammatic notation for object-oriented models. It allows to capture the properties of a system as a set of objects connected by channels. An object,

in turn, is characterised by its state space and the services it provides to the outside world. Hardware synthesis can be performed on a system partition all objects and channels of which are static. Synthesis transforms objects into hardware components with an internal data store, resources for the execution of services, and ports for communication with the environment. Channels are synthesized into a corresponding connection structure, instantiating arbitration components where necessary.

The functionality of services is not covered by the meta-model. For the purpose of this work, it is sufficient to assume that dynamic aspects are specified as sequential algorithms in some programming or hardware description language. These algorithms can be processed by subsequent high-level synthesis. The use of Objective VHDL, an object-oriented VHDL dialect, for the concrete specification of synthesis models—including dynamic aspects—has been explained and demonstrated.

Polymorphism and dynamic binding trade in greater modelling flexibility for an implementation overhead. In software, primarily the execution performance is affected. The hardware implementation of respective mechanisms is prone to a circuit size overhead for implementing unnecessary state bits and services that are never invoked. To avoid this, we have developed data flow analysis techniques for Objective VHDL models. Their implementation runs at an acceptable speed and helps reduce overhead significantly.

Generation of VHDL code allows to utilise high level synthesis tools for further processing of the design. Particular attention has been given to the VHDL implementation of an optimized bit level encoding and efficient communication code under consideration of the synthesizable language subset and tool capabilities.

The results of our design experiments, systematically covering the different aspects of object-orientation, show that hardware synthesis of digital circuits from object-oriented specifications is feasible and can be achieved by following the path described in this work. To make the approach practical for larger designs, it will be necessary to automate the generation of communication signals and protocol code, which currently still requires manual interaction. It will be desirable to specify constraints in the original, object-oriented model, and to provide an automatic translation into constraints on the synthesis of the VHDL model. Parallelizing techniques such as pipelining should be implemented at the object-oriented level since they cannot be delegated to high level synthesis effectively. Further research could address the analysis and optimization of inter-object communication, which might allow resource sharing between concurrent objects and eliminate the need for arbitration.

Appendix A: List of Acronyms

ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction Set Processor
ASSP	Application Specific Standard Product
BC	(Synopsys) Behavioral Compiler
CASE	Computer Aided Software Engineering
COOP	Concurrent Object-Oriented Programming
DC	(Synopsys) Design Compiler
DFA	Data Flow Analysis
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPI	Fixed Point Iteration
FSM	Finite State Machine
HDL	Hardware Description Language
HLS	High Level Synthesis
HRT-HOOD	Hard Real-Time Hierarchical Object-Oriented Design
HW	Hardware
IEEE	Institute of Electrical and Electronics Engineers
INSYDE	INtegrated SYstem DEvelopment (EU funded project)
I/O	Input / Output
IP	Intellectual Property
LIFO	Last In First Out
LRM	Language Reference Manual (of VHDL)

MOOSE	Model based Object-Oriented Systems Engineering
OMT	Object Modeling Technique
OO	Object-Oriented, Object-Orientation
OOA	Object-Oriented Analysis
OOA/D	Object-Oriented Analysis and Design
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
OO-HDL	Object-Oriented Hardware Description Language
OO-VHDL	Object-Oriented VHDL
RAM	Random Access Memory
RASSP	Rapid prototyping of Application Specific Signal Processors (US funded project)
REQUEST	REuse and QUality ESTimation (EU funded project)
RMI	Remote Method Invocation
ROOM	Real-time Object-Oriented Modelling
RTL	Register Transfer Level
SIA	Semiconductors Industry Association
SQUASH	Static QUality Assurance for Software and Hardware (EU funded project)
SRAM	Static Random Access Memory
SUAVE	Savant and University of Adelaide VHDL Extensions
SW	Software
UML	Unified Modeling Language
VHDL	Very High Speed Integrated Circuits Hardware Description Language
VLIW	Very Long Instruction Word
VSI	Virtual Socket Interface
VSIA	Virtual Socket Interface Alliance
XNF	Xilinx Netlist Format

Appendix B: List of Symbols

$b(c)$	number of bits for binary encoding of the state space of class c
c	class (incl. derived class) type variable of the meta-model
C	class (incl. derived class) type identifier
C^c	constraint vector of concurrent polymorphic objects
C^s	constraint vector of sequential polymorphic objects
d	derived class type variable of the meta-model
D	derived class type identifier
D^{in}, D^{out} etc.	definition vectors
$Defs$	set of all definitions (in data flow analysis)
e	entry vertex
E	set of edges
enc_c	encoding of the state space of class c
G	graph in data flow analysis / fixed point iteration
$init_c$	initial state of an object of class c
I_c	input parameter space defined by methods of class c
$k(method)$	number of input parameters of *method*
$l(method)$	number of output parameters of *method*
lb	lower bound of the index range of an array
$m(c)$	number of methods of class c
$method_{c,i}$	i-th method of class c
M_c	set of all methods of class c
$next$	state transition function
$\mathbb{N}$	the set of natural numbers, inclusive of 0
O_c	output parameter space defined by methods of class c

out	output function
$p^{in}_{c,\,i,\,j}$	j-th input parameter of the i-th method of class c
$pos_t(i)$	start position of the i-th element in the state bit-vector of type t
P	Parent class type identifier
q_c	state of an object of class c
q_{ext}	state extension of an object of a derived class w.r.t. the state inherited from its parent class
R	root class of a polymorphic object
R_{poly}	root class and derived classes
S_t	state space defined by type t
t	type variable of the meta-model
$t^{in}_{c,\,i,\,j}$	type of the j-th input parameter of the i-th method of class c
T_i , $T(v)$	set of types of object no. i, of definition represented by vertex v
$Types$	set of all types (in chapter 6 restricted to class types)
ub	upper bound of the index range of an array
v	vertex
V	set of vertices
$Values$	set of all values
$x_{(2)}$	binary (unsigned or twos-complement) encoding of value x
$\varnothing$	empty set, empty definition vector
$\bot$	bottom element of a lattice
$\top$	top element of a lattice
$\wp(.)$	power set
$\mu(.)$	edge value (constraint) function of graph G
Σ	object system
ζ	set of all channels
Ω	set of all objects
Ω^{c}_{poly}	set of concurrently used polymorphic objects
Ω^{s}_{poly}	set of sequentially used polymorphic objects
ε	the empty word

Appendix C: Syntax Summary

This appendix lists in alphabetical order the new grammar productions of Objective VHDL to the extent relevant to this work. The presentation takes into account the suggested addition of ports to class declarations. The complete language syntax is defined by the rules on type-based object-orientation presented here, additional rules for entity-based inheritance which can be found in [101], and the rules from the VHDL language reference manual (LRM) [74]. If a rule found here defines the same non-terminal as a VHDL rule, it replaces the VHDL rule in Objective VHDL. The new grammar productions are described under application of the variant of the Backus-Naur form used in the VHDL LRM.

class_attribute_declaration ::=
    class attribute identifier : subtype_indication [:= expression] ;

class_body_common_declarative_item ::=
      type_declaration
    | subtype_declaration
    | constant_declaration
    | subprogram_declaration
    | subprogram_body
    | alias_declaration
    | use_clause

class_body_declarative_item ::=
      class_attribute_declaration
    | class_body_common_declarative_item
    | class_body_object_configuration

class_body_object_configuration ::=
    for object_specification { , object_specification }
        { class_body_common_declarative_item }
    end for ;

class_type_body ::=
    class body
        { class_body_declarative_item }
    end class body [*class_type*_simple_name]

class_type_common_declarative_item ::=
      type_declaration
    | subtype_declaration
    | constant_declaration
    | subprogram_declaration
    | alias_declaration
    | use_clause

class_type_declaration ::=
    [**abstract**] **class**
        [*formal*_generic_clause]
        [*formal*_port_clause]
        { class_type_declarative_item }
    end class [*class_type*_simple_name]

class_type_declarative_item ::=
      class_attribute_declaration
    | class_type_common_declarative_item
    | class_type_object_configuration

class_type_definition ::=
      class_type_declaration
    | derived_class_type_declaration
    | class_type_body

class_type_object_configuration ::=
    for object_specification { , object_specification }
        { class_type_common_declarative_item }
    end for ;

derived_class_type_declaration ::=
    new [**abstract**] **class** *class_type*_name **with**
        [*formal*_generic_clause]
        [*formal*_port_clause]
        { class_type_declarative_item }
    end class [*class_type*_simple_name]

function_call ::=
    *function*_name [(actual_parameter_part)]
  | prefix . *class_function*_name [(actual_parameter_part)]

object_specification ::=
  signal | **variable** | **constant**

procedure_call ::=
    *procedure*_name [(actual_parameter_part)]
  | prefix . *class_procedure*_name [(actual_parameter_part)]

subtype_indication ::=
    [resolution_function_name] type_mark [constraint]
  | [resolution_function_name] *class*_type_mark
      [generic_map_aspect]
      [port_map_aspect]

type_definition ::=
  scalar_type_definition
  composite_type_definition
  access_type_definition
  file_type_definition
  class_type_definition

Appendix D: Synthesis Subset

This appendix lists the synthesizable subset of Objective VHDL as far as relevant to this work. Hence, the presentation is limited to the synthesizability of class types and their use. A full definition, taking entity-based inheritance into account, too, can be found in [132]. The rules presented here are to be understood as an addition to those that describe the language subset accepted by the VHDL synthesis tool used. The terminology and style of our subset definition follows the approach of the RTL synthesis subset (IEEE approved standard 1076.6) [75]. An underlined construct is ignored during synthesis. A construct that is stuck through is rejected by the synthesizer (not supported). If this limitation may be lifted in the future, we speak of “currently not supported“. The other constructs are supported.

1 Class types

```
type_definition ::=
      scalar_type_definition
    | composite_type_definition
    | access_type_definition
    | file_type_definition
    | class_type_definition    -- new w.r.t. VHDL

class_type_definition ::=
      class_type_declaration
    | derived_class_type_declaration
    | class_type_body          -- new w.r.t. VHDL

scalar, composite, access, and file type definition as in VHDL/1076.6
```

Supported:

- class type declaration (currently only in package declaration)
- derived class type declaration (currently only in package declaration)
- class type body (currently only in package body)

Currently not supported:
- class type declaration and derived class type declaration in an enclosing scope that is not a package declaration
- class type body in an enclosing scope that is not a package body

1.2 Declaration of class types

```
class_type_declaration ::=
    [ abstract ] class
        [ formal_generic_clause ]
        { class_type_declarative_item }
    end class [ class_type_simple_name ] ;

derived_class_type_declaration ::=
    new [ abstract ] class class_type_name with
        [ formal_generic_clause ]
        { class_type_declarative_item }
    end class [ class_type_simple_name ] ;
```

Supported:
- class type declaration, including abstract class type
- derived class type declaration, including abstract derived class type
- generic clause (as in VHDL / 1076.6)
- class type declarative item

1.2.1 Class type declarative item

```
class_type_declarative_item ::=
      class_attribute_declaration
    | class_type_common_declarative_item
    | class_type_object_configuration

class_type_common_declarative_item ::=
      type_declaration
    | subtype_declaration
    | constant_declaration
    | subprogram_declaration
    | alias_declaration
    | use_clause
```

type, subtype, constant, subprogram, alias declaration and use clause as in 1076.6

Supported:
- class type declarative item
- class attribute declaration
- class type common declarative item
- class type object configuration
- type, subtype, constant (non-deferred), subprogram declaration
- use clause (of all items or a particular item of a package)

Ignored:
- alias declaration

A use clause must only reference the selected name of a package (which may in turn reference all, or a particular item_name within the package).

A constant declaration must include the initial value expression, that is, deferred constants are not supported.

1.2.2 Class attribute declaration

```
class_attribute_declaration ::=
    class attribute identifier : subtype_indication [ := expression ] ;
```

Supported:
- class attribute declaration

Currently not supported:
- class attribute declaration without initial value expression

The subtype indication must not denote an access or file (sub)type, nor a composite type which contains an element of an access or file (sub)type.

The initial value of a class attribute is assumed as undefined by synthesis. The user must describe reset behaviour explicitly to obtain initialisation.

Note: this grammar production may in the future be extended to allow an identifier list. The identifier list will be supported for synthesis.

Note: although it is ignored, there must currently be an initial value expression due to a compiler bug. We expect to remove this problem in the next tool revision.

1.2.3 Class type object configuration

```
class_type_object_configuration ::=
    for object_specification { , object_specification }
        { class_type_common_declarative_item }
    end for ;

    object_specification ::= signal | variable | constant
```

Supported:
- class type object configuration
- object specification

Currently not supported:
- multiple object specifications

A function declaration is currently not supported in a class type object configuration whose object specification is **variable**.

1.3 Declaration of class bodies

```
class_type_body ::=
    class body
        { class_body_declarative_item }
    end class body [ class_type_simple_name ]
```

Supported:
- class type body

1.3.1 Class body declarative item

```
class_body_declarative_item ::=
      class_attribute_declaration
    | class_body_common_declarative_item
    | class_body_object_configuration
    | class_body_entity_configuration
```

```
class_body_common_declarative_item ::=
      type_declaration
    | subtype_declaration
    | constant_declaration
    | subprogram_declaration
    | subprogram_body
    | alias_declaration
    | use_clause
```

Supported:
- class body declarative item
- class body common declarative item

Ignored:
- alias declaration

Not supported:
- class body entity configuration

Currently not supported:
- class attribute declaration

A use clause shall only reference the selected name of a package (which may in turn reference all, or a particular item_name within the package).

A constant declaration must include the initial value expression, that is, deferred constants are not supported.

Note: the class body entity configuration may become obsolete in a future revision of Objective VHDL.

1.3.2 Class body object configuration

```
class_body_object_configuration ::=
    for object_specification { , object_specification }
        { class_body_common_declarative_item }
    end for ;

object_specification ::= signal | variable | constant   -- same as above
```

Supported:
- class body object configuration
- object specification

Currently not supported:
- multiple object specifications

A function or function body is currently not supported if declared in a class body object configuration whose object specification is **variable**.

1.3.3 Class body entity configuration

```
class_body_entity_configuration ::=
    for entity entity_name
        { class_body_entity_configuration_declarative_item }
    end for ;
```

Not supported:
- class body entity configuration

1.4 Use of class types

```
subtype_indication ::=
      [ resolution_function_name ] type_mark [ constraint ]-- VHDL
    | [ resolution_function_name ] type_mark generic_map_aspect-- new
```

Supported:
- subtype indication (particularly also if the type mark denotes a class type or derived class type or class-wide type)
- generic map aspect if the enclosing scope of the subtype indication is a package declaration

Ignored:
- user-defined resolution functions

Currently not supported:
- generic map aspect in an enclosing scope that is not a package declaration

1.5 Use of class instances

```
procedure_call ::=
      procedure_name [ ( actual_parameter_part ) ]-- VHDL
    | prefix . class_procedure_name [ ( actual_parameter_part ) ]-- new

function_call ::=
      function_name [ ( actual_parameter_part ) ]-- VHDL
    | prefix . class_function_name [ ( actual_parameter_part ) ]-- new
```

Supported:
- procedure call (particularly also the prefixed version)
- function call (particularly also the prefixed version)
- prefix that is a simple name

Currently not supported:
- prefix that is an indexed name, a selected name, or a function call
- prefix THIS

2 Predefined language environment

2.1 Predefined type

Supported:
- *universal_tag*

2.2 Predefined attributes

Supported:
- the class-wide type T'CLASS (where T is a class type or derived class type)
- T'TAG (where T is a class type or derived class type)
- S'TAG (where S is a signal of class type, derived class type, or class-wide type)
- V'TAG (where V is a variable of class type, derived class type, or class-wide type)

- C'TAG (where C is a constant of class type, derived class type, or class-wide type)
- standard VHDL attributes that are supported according to 1076.6 and applicable to Objective VHDL items

2.3 Predefined functions

Supported:

- the relational operators (=, /=, <, >, <=, >=) for the predefined type *universal_tag*

The use of these operators with two variable tags is currently not supported. A variable tag results from the use of the TAG attribute with a signal or variable of a class-wide type.

Appendix E: OOSYN library

This appendix describes the contents of the Objective VHDL library OOSYN which contains packages with declarations relevant to synthesis and VHDL code generation.

```
library IEEE;
use IEEE.std_logic_1164.all;

package SYN_SCALAR_TYPES is

    -- auxiliary functions

    function INTLOG2( ARG : Natural ) return Natural;

    type INTEGER_VECTOR is array( Natural range <> ) of Integer;
    function INTMAX( ARG : INTEGER_VECTOR ) return Integer;

    -- conversion of integer / natural types and subtypes

    function INT_TO_STDV( ARG : Integer; SIZE : Natural )
        return Std_logic_vector;

    function NAT_TO_STDV( ARG : Natural; SIZE : Natural )
        return Std_logic_vector;

    function STDV_TO_INT( ARG : Std_logic_vector ) return Integer;

    function STDV_TO_NAT( ARG : Std_logic_vector ) return Natural;

    -- conversion of boolean and bit type

    function BOOL_TO_STDL( ARG : Boolean ) return Std_LOGIC;

    function STDL_TO_BOOL( ARG : Std_logic ) return Boolean;

end SYN_SCALAR_TYPES;

-- package implementation for Synopsys

use IEEE.std_logic_arith.all;

package body SYN_SCALAR_TYPES is
```

```
-- auxiliary functions

function RECURSE_INTLOG2( LOG, POWER, ARG : Natural )
    return Natural;

function INTLOG2( ARG : Natural ) return Natural is
begin
    -- LOOPS NOT SUPPORTED FOR SYNOPSYS ELABORATE -S
    -- HENCE RECURSIVE IMPLEMENTATION
    return RECURSE_INTLOG2( 0, 1, ARG );
end;

function RECURSE_INTLOG2( LOG, POWER, ARG : Natural )
    return Natural is
begin
    if POWER < ARG then
        return RECURSE_INTLOG2( LOG+1, POWER*2, ARG );
    else
        return LOG;
    end if;
end;

function INTMAX( ARG : INTEGER_VECTOR ) return Integer is
    variable MAX : INTEGER := INTEGER'LEFT;
begin
    -- Assertion: ARG is not an empty vector (null range)
    -- This is unchecked as the function is only for tool use.
    for I in ARG'RANGE loop
        if ARG(I) > MAX then
           MAX := ARG(I);
        end if;
    end loop;
    return MAX;
end;

function INT_TO_STDV( ARG : Integer; SIZE : Natural )
    return Std_logic_vector is
begin
    return conv_std_logic_vector( conv_signed( ARG, SIZE ), SIZE );
end;

function NAT_TO_STDV( ARG : Natural; SIZE : Natural )
    return Std_logic_vector is
begin
```

```
        return conv_std_logic_vector(conv_unsigned(ARG, SIZE),SIZE);
    end;

    function STDV_TO_INT( ARG : Std_logic_vector ) return Integer is
    begin
        return IEEE.STD_LOGIC_SIGNED.CONV_INTEGER( ARG );
    end;

    function STDV_TO_NAT( ARG : Std_logic_vector ) return Natural is
    begin
        return IEEE.STD_LOGIC_UNSIGNED.CONV_INTEGER(ARG);
    end;

    function BOOL_TO_STDL( ARG : Boolean ) return Std_logic is
    begin
        if ARG then
            return '1';
        else
            return '0';
        end if;
    end;

    function STDL_TO_BOOL( ARG : Std_logic ) return Boolean is
    begin
        return ARG = '1';
    end;

end SYN_SCALAR_TYPES;


library IEEE;
use IEEE.std_logic_1164.all;

package SYN_COMPOSITE_TYPES is

    type PARAM_VECTOR is array( Positive range <> ) of Natural;

    function EXTRACT_ELEMENT(
        constant A : Std_logic_vector;
        constant INDEX  : PARAM_VECTOR;
        constant SIZE   : PARAM_VECTOR;
        constant BASE   : PARAM_VECTOR;
        constant OFFSET : PARAM_VECTOR )
        return Std_logic_vector;
```

```
    procedure ASSIGN_ELEMENT(
        variable A : inout Std_logic_vector;
        constant INDEX  : PARAM_VECTOR;
        constant SIZE   : PARAM_VECTOR;
        constant BASE   : PARAM_VECTOR;
        constant OFFSET : PARAM_VECTOR;
        constant V : STD_LOGIC_VECTOR );

end SYN_COMPOSITE_TYPES;

use WORK.SYN_SCALAR_TYPES.all;

package body SYN_COMPOSITE_TYPES is

    function EXTRACT_RECURSE(
        constant A : Std_logic_vector;
        constant INDEX  : PARAM_VECTOR;
        constant SIZE   : PARAM_VECTOR;
        constant BASE   : PARAM_VECTOR;
        constant OFFSET : PARAM_VECTOR;
        constant STEPS  : Natural )
        return Std_logic_vector;

    function EXTRACT_ELEMENT(
        constant A : Std_logic_vector;
        constant INDEX  : PARAM_VECTOR;
        constant SIZE   : PARAM_VECTOR;
        constant BASE   : PARAM_VECTOR;
        constant OFFSET : PARAM_VECTOR )
        return Std_logic_vector
    is
        constant STEPS : Natural
            := intlog2( (A'RIGHT-A'LEFT+1) / SIZE(SIZE'LEFT) );
    begin
        return EXTRACT_RECURSE(
                    A, INDEX, SIZE, BASE, OFFSET, STEPS );
    end;

    function EXTRACT_RECURSE(
        constant A : Std_logic_vector;
        constant INDEX  : PARAM_VECTOR;
        constant SIZE   : PARAM_VECTOR;
        constant BASE   : PARAM_VECTOR;
        constant OFFSET : PARAM_VECTOR;
        constant STEPS  : Natural )
```

```
    return STD_LOGIC_VECTOR
is
    constant I : NATURAL := OFFSET'LEFT;
    constant J : NATURAL := BASE'LEFT;
    constant PIVOT : NATURAL
        := 2**(intlog2((A'RIGHT-A'LEFT+1) / SIZE(I))) / 2;
    variable NEW_BASE : PARAM_VECTOR(BASE'RANGE);
begin
    NEW_BASE := BASE;
    if STEPS = 0 then
        if I = OFFSET'RIGHT then
            return A;
        else
            return EXTRACT_ELEMENT(
                        A(A'LEFT + OFFSET(I) to A'RIGHT),
                        INDEX( I+1 to INDEX'RIGHT),
                        SIZE(  I+1 to SIZE'RIGHT ),
                        BASE(  J+1 to BASE'RIGHT ),
                        OFFSET(I+1 to OFFSET'RIGHT) );
        end if;
    elsif (INDEX(I) mod (2**STEPS)) / (2**(STEPS-1)) = 0 then
        return EXTRACT_ELEMENT(
                    A(A'LEFT to A'LEFT + PIVOT * SIZE(I) - 1),
                    INDEX,
                    SIZE,
                    BASE,
                    OFFSET );
    else
        NEW_BASE( J ) := BASE( J ) + PIVOT;
        return EXTRACT_ELEMENT(
                    A(A'LEFT + PIVOT * SIZE(I) to A'RIGHT),
                    INDEX,
                    SIZE,
                    NEW_BASE,
                    OFFSET );
    end if;
end;

procedure ASSIGN_RECURSE(
    variable A : inout Std_logic_vector;
    constant INDEX  : PARAM_VECTOR;
    constant SIZE   : PARAM_VECTOR;
    constant BASE   : PARAM_VECTOR;
```

```
    constant OFFSET : PARAM_VECTOR;
    constant STEPS  : NATURAL;
    constant V : Std_logic_vector );

procedure ASSIGN_ELEMENT(
    variable A : inout Std_logic_vector;
    constant INDEX  : PARAM_VECTOR;
    constant SIZE   : PARAM_VECTOR;
    constant BASE   : PARAM_VECTOR;
    constant OFFSET : PARAM_VECTOR;
    constant V : Std_logic_vector )
is
    constant STEPS : Natural
            := intlog2( (A'RIGHT-A'LEFT+1) / SIZE(SIZE'LEFT) );
begin
    ASSIGN_RECURSE(
        A, INDEX, SIZE, BASE, OFFSET, STEPS, V );
end;

procedure ASSIGN_RECURSE(
    variable A : inout Std_logic_vector;
    constant INDEX  : PARAM_VECTOR;
    constant SIZE   : PARAM_VECTOR;
    constant BASE   : PARAM_VECTOR;
    constant OFFSET : PARAM_VECTOR;
    constant STEPS  : Natural;
    constant V : Std_logic_vector )
is
    constant I : NATURAL := OFFSET'LEFT;
    constant J : NATURAL := BASE'LEFT;
    constant PIVOT : Natural
            := 2**(intlog2((A'RIGHT-A'LEFT+1) / SIZE(I))) / 2;
    variable NEW_BASE : PARAM_VECTOR(BASE'RANGE);
begin
    NEW_BASE := BASE;
    if STEPS = 0 then
        if I = OFFSET'RIGHT then
            A := V;
        else
            ASSIGN_ELEMENT(
                A(A'LEFT + OFFSET(I) to A'RIGHT),
                INDEX( I+1 to INDEX'RIGHT),
                SIZE(  I+1 to SIZE'RIGHT),
```

```
                    BASE(  J+1 to BASE'RIGHT),
                    OFFSET(I+1 to OFFSET'RIGHT),
                    V );
            end if;
        elsif (INDEX(I) mod (2**STEPS)) / (2**(STEPS-1)) = 0 then
            ASSIGN_ELEMENT(
                A(A'LEFT to A'LEFT + PIVOT * SIZE(I) - 1),
                INDEX,
                SIZE,
                BASE,
                OFFSET,
                V );
        else
            NEW_BASE( J ) := BASE( J ) + PIVOT;
            ASSIGN_ELEMENT(
                A(A'LEFT + PIVOT * SIZE(I) to A'RIGHT),
                INDEX,
                SIZE,
                NEW_BASE,
                OFFSET,
                V );
        end if;
    end;

end SYN_COMPOSITE_TYPES;
```

Appendix F: Scheduler Implementation

This appendix lists the source code of the schedulers currently implemented.

```
library IEEE;
use IEEE.std_logic_1164.all;

use WORK.SYN_ARRAY.all;

entity SCHEDULER_STATE_ONLY is

    generic(
        NO_CLIENTS : Positive;
        REQ_BITS   : Positive );

    port(
        CLK        : in Std_logic;
        RESET      : in Std_logic;
        S_RDY      : in Std_logic_vector(0 to 2**REQ_BITS-1);
        C_REQ      : in Std_logic_vector(0 to NO_CLIENTS*REQ_BITS-1);
        S_DONE     : in Std_logic;
        GRANT      : out Integer range 0 to NO_CLIENTS );

end;

architecture STATIC_PRIORITY of SCHEDULER_STATE_ONLY is

    signal SEL      : Integer range 0 to NO_CLIENTS;
    signal PAST_SEL : Integer range 0 to NO_CLIENTS;

begin

    REG : process( CLK )
    begin
        if CLK'EVENT and CLK = '1' then
            if RESET = '1' then
                PAST_SEL <= 0;
            else
                PAST_SEL <= SEL;
            end if;
```

```
        end if;
    end process;

    COMB : process( PAST_SEL, S_DONE, C_REQ, S_RDY )
        variable NXT : INTEGER range 0 to NO_CLIENTS;
    begin
        if S_DONE = '1' then
            for I in NO_CLIENTS downto 0 loop
                NXT := I;
                exit when I = 0;
                exit when S_RDY(
                    IEEE.STD_LOGIC_UNSIGNED.CONV_INTEGER(
                        EXTRACT_ELEMENT( C_REQ, (NXT-1, 0),
                                (REQ_BITS, REQ_BITS), (0,0), (0,0) )
                    ) ) = '1';
            end loop;
            GRANT <= NXT;
            SEL <= NXT;
        else
            GRANT <= PAST_SEL;
            SEL <= PAST_SEL;
        end if;
    end process;

end STATIC_PRIORITY;

architecture ROUND_ROBIN of SCHEDULER_STATE_ONLY is

    signal SEL          : Integer range 0 to NO_CLIENTS;
    signal PAST_SEL     : Integer range 0 to NO_CLIENTS;

begin

    REG : process( CLK )
    begin
        if CLK'EVENT and CLK = '1' then
            if RESET = '1' then
                PAST_SEL <= 0;
            else
                PAST_SEL <= SEL;
            end if;
        end if;
    end process;

    COMB : process( PAST_SEL, S_DONE, C_REQ, S_RDY )
        variable NXT : INTEGER range 0 to NO_CLIENTS;
```

```
    begin
        if S_DONE = '1' then
            for I in 0 to NO_CLIENTS loop
                if I = NO_CLIENTS then
                    NXT := 0;
                    exit;
                end if;
                if PAST_SEL+1+I < NO_CLIENTS+1 then
                    NXT := PAST_SEL+1+I;
                else
                    NXT := PAST_SEL+1+I-NO_CLIENTS;
                end if;
                exit when
                    S_RDY(
                        IEEE.STD_LOGIC_UNSIGNED.CONV_INTEGER(
                            EXTRACT_ELEMENT( C_REQ, (NXT-1, 0),
                                (REQ_BITS, REQ_BITS), (0,0), (0,0) )
                        ) ) = '1';
            end loop;
            GRANT <= NXT;
            SEL <= NXT;
        else
            GRANT <= PAST_SEL;
            SEL <= PAST_SEL;
        end if;
    end process;

end ROUND_ROBIN;

architecture MODIFIED_ROUND_ROBIN of
    SCHEDULER_STATE_ONLY is

    signal SEL        : Integer range 0 to NO_CLIENTS;
    signal PAST_SEL   : Integer range 0 to NO_CLIENTS;
    signal START      : Integer range 0 to NO_CLIENTS;
    signal PAST_START : Integer range 0 to NO_CLIENTS;

begin

    REG : process( CLK )
    begin
        if CLK'EVENT and CLK = '1' then
            if RESET = '1' then
                PAST_SEL <= 0;
                PAST_START <= 0;
```

```
            else
                PAST_SEL <= SEL;
                PAST_START <= START;
            end if;
        end if;
    end process;

    COMB : process( PAST_SEL, PAST_START, S_DONE, C_REQ,
                    S_RDY )
        variable NXT : Integer range 0 to NO_CLIENTS;
        variable NXT_START : Integer range 0 to NO_CLIENTS;
    begin
        if S_DONE = '1' then
            for I in 0 to NO_CLIENTS loop
                if I = NO_CLIENTS then
                    NXT := 0;
                    exit;
                end if;
                if PAST_START+1+I < NO_CLIENTS+1 then
                    NXT := PAST_START+1+I;
                else
                    NXT := PAST_START+1+I-NO_CLIENTS;
                end if;
                exit when
                    S_RDY(
                        IEEE.STD_LOGIC_UNSIGNED.CONV_INTEGER(
                            EXTRACT_ELEMENT( C_REQ, (NXT-1, 0),
                                (REQ_BITS, REQ_BITS), (0,0), (0,0) )
                            ) ) = '1';
            end loop;
            for I in 0 to NO_CLIENTS-1 loop
                if PAST_START+1+I < NO_CLIENTS+1 then
                    NXT_START := PAST_START+1+I;
                else
                    NXT_START := PAST_START+1+I-NO_CLIENTS;
                end if;
                exit when
                    EXTRACT_ELEMENT( C_REQ, (NXT_START-1,0),
                        (REQ_BITS,REQ_BITS), (0,0), (0,0) ) /= "00";
            end loop;
            GRANT <= NXT;
            SEL <= NXT;
            if NXT = NXT_START then
```

```
                START <= NXT_START;
            else
                --START <= (NXT_START - 1) mod NO_CLIENTS;
                assert NXT_START /= 0
                    report "NXT_START is 0" severity failure;
                if NXT_START = 1 then
                    START <= NO_CLIENTS;
                else
                    START <= NXT_START-1;
                end if;
            end if;
        else
            GRANT <= PAST_SEL;
            SEL <= PAST_SEL;
            START <= PAST_START;
        end if;
    end process;

end MODIFIED_ROUND_ROBIN;
```

References

[1] K. Agsteiner, D. Monjau, S. Schulze. *Object-Oriented High-Level Modeling of System Components for the Generation of VHDL Code*. Proc. EURO-DAC with EURO-VHDL, 1995.

[2] K. Agsteiner, D. Monjau, S. Schulze. *Ein objektorientiertes Modell als Basis für Spezifikation, Prototyping, Implementierung und Wiederverwendung* (German). Proc. GI/ITG/GMM Workshop Hardwarebeschreibungssprachen und Modellierungsparadigmen, 1997.

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

[4] A. Allara, A. Balboni, M. Bombana, M. Mastretti, J. R. Prieto, P. Plaza, J. Schaaf. *Productivity gains through re-use and quality improvements of HW models*. Proc. 2nd Workshop on Libraries, Component Modeling and Quality Assurance, Toledo, 1997.

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

[6] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[7] K. Arnold, J. Gosling. *The Java programming language.* Addison-Wesley, 1996.

[8] G. Arnout. *C for System Level Design*. Proc. Design, Automation and Test in Europe (DATE), 1999.

[9] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, 1996.

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

[11] P. J. Ashenden, P. A. Wilsey, D. E. Martin. *Reuse Through Genericity in SUAVE*. Proc. VHDL International Users' Forum (VIUF, Fall Conference), 1997.

[12] P. J. Ashenden, P. A. Wilsey. *A Comparison of Alternative Extensions for Data Modeling in VHDL*. Proc. Hawaii International Conference

on System Sciences (HICSS), 1998.

[13] P. J. Ashenden, P. A. Wilsey. *Considerations on System-Level Behavioural and Structural Modeling Extensions to VHDL*. Proc. VHDL International Users' Forum (VIUF, Spring Conference), 1998.

[14] P. J. Ashenden, P. A. Wilsey, D. E. Martin. *SUAVE: Extending VHDL to Improve Modeling Support.* IEEE Design & Test of Computers, vol. 15, no. 2, 1998.

[15] P. J. Ashenden, P. A. Wilsey. *Extensions to VHDL for Abstraction of Concurrency and Communication*. Proc. 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 1998.

[16] P. J. Ashenden, P. A. Wilsey, D. E. Martin. *SUAVE: Object-Oriented and Genericity Extensions to VHDL for High-Level Modeling*. Proc. Forum on Design Languages (FDL), 1998.

[17] P. J. Ashenden, P. A. Wilsey, W. Nebel, M. Radetzki, W. Putzke-Röming, G. D. Peterson. *SUAVE and Objective VHDL: Object-Oriented Extensions to VHDL*. Proc. Forum on Design Languages (FDL), 1999.

[18] P. J. Ashenden. *Overview of SUAVE Language Features.* Proc. Forum on Design Languages (FDL), 1999.

[19] P. J. Ashenden, M. Radetzki. *Comparison of SUAVE and Objective VHDL Language Features*. Proc. Forum on Design Languages (FDL), 1999.

[20] C. Barna, W. Rosenstiel. *Description and Classification of VHDL Objects in the Reuse Management System*. Proc. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, 1999.

[21] C. Barna, W. Rosenstiel. *Object-Oriented Reuse Methodology for VHDL*. Proc. Design, Automation and Test in Europe (DATE), 1999.

[22] J. Barnes. *Programming in Ada95*. Addison-Wesley, 1995.

[23] K. Bartleson. *A New Standard for System-Level Design.* Synopsys Inc., 1999.

[24] J. Benzakki, B. Djafri. *Object Oriented Extensions to VHDL—The LaMI proposal*. Proc. Conf. on Computer Hardware Description Languages (CHDL), 1997.

[25] J. Böttger, W. Ecker, M. Mrva. *Klassifikation von objektorientierten VHDL-Erweiterungen* (German). Proc. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, 1998.

[26] M. Bombana. *HDLs in Industry.* Proc. Forum on Design Languages (FDL), 1999.

[27] G. Booch. *Object Oriented Design*. Benjamin / Cummings Publishing, 1991.

[28] G. Booch. *The Best of Booch*. SIGS Books and Multimedia, New York, 1996.

[29] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

[30] G. Bracha, W. Cook. *Mixin-based Inheritance*. Proc. ECOOP/ OOPSLA'90. Sigplan Notices, vol. 25, no. 10, 1990.

[31] P. Brinch Hansen. *Structured Multiprogramming*. Communications of the ACM, vol. 15, no. 7, 1972, pp. 574-578.

[32] A. Burns, B. Dobbing, G. Romanski. *The Ravenscar Tasking Profile for High Integrity Real-Time Programs*. In: Lars Asplund (Ed.), Proc. Reliable Software Technologies—Ada-Europe '98. Lecture Notes in Computer Science 1411, Springer, 1998.

[33] D. Cabanis, S. Medhat, N. Weavers. *Classification-Orientation for VHDL: A Specification*. Proc. VHDL International Users' Forum (VIUF, Spring Conference), 1996.

[34] D. Cabanis, S. Medhat. *Object-Oriented Extensions to VHDL: The Classification Orientation*. Proc. VHDL User Forum in Europe (VFE), 1996.

[35] R. H. Campbell, A. N. Habermann. *The specification of process synchronization by path expressions*. In: Lecture Notes in Computer Science, 16. Springer-Verlag, 1974, pp. 89-102.

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

[37] B. M. Covnot, D. W. Hurst, S. Swamy. *OO-VHDL: An Object-Oriented VHDL*. Proc. VHDL International Users' Forum (VIUF), 1994.

[38] CynApps, Inc. *Cynlib, CynApps Class Library.* Information sheet, 1999.

[39] J. L. da Silva, C. Ykman-Couvreur, G. de Jong. *Matisse: a concurrent and object-oriented system specification language*. Proc. IFIP International Conference on VLSI, 1997.

[40] J. L. da Silva, C. Ykman-Couvreur, G. de Jong, B. Lin, H. De Man. *A System Design Methodology for Telecommunication Network Applications*. Proc. Great Lakes Symposium on VLSI, 1997.

[41] J. L. da Silva et al. *Efficient System Exploration and Synthesis of Applications with Dynamic Data Storage and Intensive Data Transfer.* Proc. Design Automation Conference (DAC), 1998.

[42] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[43] E. W. Dijkstra. *Hierarchical Ordering of Sequential Processes*. Acta Informatica **1** (1971), pp. 115-138.

[44] B. P. Douglas. *Real-Time UML*. Addison-Wesley, 1997.

[45] W. Ecker. *An Object-Oriented View of Structural VHDL Description*. Proc. VHDL International Users' Forum (Spring Conference), 1996.

[46] W. Ecker. *Neue Verfahren für den Entwurf digitaler Systeme mit Hardwarebeschreibungssprachen* (German). Dissertation, Shaker Verlag, Aachen, 1996.

[47] P. Eles, K. Kuchcinski, Z. Peng, M. Minea. *Synthesis of VHDL Concurrent Processes*. Proc. EURO-DAC with EURO-VHDL, 1994.

[48] R. Ernst, J. Henkel, Th. Benner. *Hardware-Software Cosynthesis for Microcontrollers*. IEEE Design & Test of Computers, vol 10, no 4, 1993.

[49] R. Ernst, Th. Benner. *Communication, Constraints and User Directives in COSYMA*. Technical report TM CY-94-2, Institute IDA, TU Braunschweig, 1994.

[50] R. Ernst. *Codesign of Embedded Systems: Status and Trends*. IEEE Design & Test of Computers, vol 15, no 2, 1998.

[51] T. Fandrey. *Objektorientierte Modellierung von Kommunikationskanälen*. Diploma thesis, Oldenburg University, 1997.

[52] M. E. Fayad, D. C. Schmidt. *Object-Oriented Application Frameworks*. Communications of the ACM, vol. 40, no. 10, 1997.

[53] S. Ferenczi. *Guarded Methods vs. Inheritance Anomaly—Inheritance Anomaly Solved by Nested Guarded Method Calls*. ACM SIGPLAN Notices, vol. 30, no. 2, 1995.

[54] M. Fowler, K. Scott. *UML distilled: applying the standard object modeling language*. Addison-Wesley, 1998.

[55] N. Francez. *Fairness*. Springer, 1986.

[56] D. D. Gajski, F. Vahid, S. Narayan, J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[57] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[58] W. Glunz. *Extensions from VHDL to VHDL++*. JESSI-AC8 report S2-

SP1-T2.4-Q3, 1991.

[59] W. Glunz, A. Pyttel, G. Venzl. *System-Level Synthesis*. In: P. Michel, U. Lauther, P. Duzy (eds): The Synthesis Approach to Digital System Design. Kluwer Academic Publishers, 1992.

[60] G. Goos, J. Hartmanis (Eds.). *The Programming Language Ada Reference Manual*. Lecture Notes in Computer Science 155, Springer-Verlag, 1983.

[61] G. Grätzer. *General Lattice Theory.* Birkhäuser, 1978.

[62] F. El Guibaly. *Design and Analysis of Arbitration Protocols*. IEEE Transactions on Computers, vol 38, no 2, 1989.

[63] R. K. Gupta. *Co-Synthesis of Hardware and Software*. Kluwer Academic Publishers, 1995.

[64] R. Helaihel, K. Olukotun. *Java as a Specification Language for Hardware-Software-Systems*. Proc. International Conference on Computer Aided Design (ICCAD), 1997.

[65] J. L. Hennessy, D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[66] C. A. R. Hoare. *Monitors: An Operating System Structuring Concept*. Communications of the ACM, vol. 17, no. 10 (1974), pp. 549-557.

[67] E. Holz et al. *INSYDE Integrated Methods for Evolving System Design—Application Guidelines*. ESPRIT Project 8641 Report, Humboldt University Berlin, 1995.

[68] HOOD Technical Group. *The HOOD Reference Manual, Release 4*. Reference HRM4-9/26/95, 1995. Available from http://www.hood.be

[69] Y.-W. Hsieh, S. P. Levitan. *Control/Data-flow Analysis for VHDL Semantic Extraction*. Proc. 4th Asia-Pacific Conference on Hardware Description Languages (APCHDL), 1997.

[70] Y.-W. Hsieh, S. P. Levitan. *Control/Data-flow Analysis for VHDL Semantic Extraction*. Journal of Information Science and Engineering 14 (1998), pp. 547-565.

[71] P.-A. Hsiung, C.-H. Chen, T.-Y. Lee, S.-J. Chen. *ICOS: An Intelligent Concurrent Object-Oriented Synthesis Methodology for Multiprocessor Systems*. ACM Transactions on Design Automation of Electronic Systems, vol. 3, no. 2, 1998, pp. 109-135.

[72] B. Hunting. *Polymorphism and Virtual Functions.* Embedded Systems Programming, July 1996.

[73] IEEE. *Standard VHDL Language Reference Manual*. IEEE Std 1076-1987, 1988.

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

1993, 1994.

[75] IEEE. *Draft Standard For VHDL Register Transfer Level Synthesis.* P1076.6 / D1.12a, 1999.

[76] I. Jacobson. *Object-oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1994.

[77] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[78] A. A. Jerraya, K. O'Brien. *SOLAR: An Intermediate Format for System-Level Modeling and Synthesis*. In: Buchenrieder, K.; Rozenblit, J. W.: Computer-Aided Software/Hardware Engineering. IEEE Press, 1994.

[79] A. A. Jerraya, H. Ding, P. Kission, M. Rahmouni. *Behavioral Synthesis and Component Reuse with VHDL*. Kluwer Academic Publishers, 1997.

[80] A. A. Jerraya, J. Mermet (Eds). *System-Level Synthesis*. NATO Science Series, Series E: Applied Sciences, vol. 357. Kluwer Academic Publishers, 1999.

[81] K. Kennedy. *A Survey of Data Flow Analysis Techniques*. In: S. S. Muchnick, N. D. Jones (Eds.): Program Flow Analysis—Theory and Applications, Prentice-Hall, 1981.

[82] G. Kildall. *A unified approach to global program optimization*. ACM Symposium on Principles of Programming Languages, 1973.

[83] W. Kirchgässner. *Ein Modell zur Analyse programminhärenter Zusicherungen*. GMD-Bericht Nr. 173, R. Oldenbourg Verlag, 1988.

[84] D. W. Knapp. *Behavioral Synthesis. Digital System Design Using the Synopsys Behavioral Compiler.* Prentice Hall, 1996.

[85] D. C. Ku, G. De Micheli. *Relative Scheduling Under Timing Constraints: Algorighms for High-Level Synthesis of Digital Circuits*. IEEE Transactions on Computer-Aided Design, vol. 11, no. 6, 1992.

[86] K. Küçükçakar, C.-T. Chen, J. Gong, W. Philipsen, Th. E. Tkacik. *Matisse: An Architectural Design Tool for Commodity ICs*. IEEE Design & Test of Computers, vol. 15, no. 2, 1998.

[87] T. Kuhn, W. Rosenstiel. *Java Based Modeling and Simulation of Digital Systems on Register Transfer Level.* Proc. Workshop on System Design Automation, 1998.

[88] T. Kuhn, U. Kebschull, W. Rosenstiel. *Domänenübergreifende Hardwarebeschreibung und Simulation mit Java* (German). Proc. GI/ITG Workshop Java und Eingebettete Systeme, 1998.

[89] T. Kuhn, W. Rosenstiel, U. Kebschull. *Object Oriented Hardware*

Modeling and Simulation Based on Java. Proc. International Workshop on IP Based Synthesis and System Design, 1998.

[90] T. Kuhn, W. Rosenstiel, U. Kebschull. *Beschreibung und Simulation von Hardware/Software-Systemen mit Java* (German). Proc. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, 1999.

[91] T. Kuhn, W. Rosenstiel, U. Kebschull. *Description and Simulation of Hardware/Software Systems with Java.* Proc. Design Automation Conference, 1999.

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

[93] LEDA S.A. *VHDL*Verilog System—User's Manual—VHDL Compiler Version 4.1*, 1997.

[94] LEDA S.A. *VHDL*Verilog System—Implementor's Guide—VHDL Intermediate Format (VIF) Version 4.1*, 1997.

[95] LEDA S.A. *VHDL*Verilog System—Implementor's Guide—OO-VHDL Extension (OO-VIF) Version A.1*, 1997.

[96] LEDA S.A. *VHDL*Verilog System—Implementor's Guide—LEDA Procedural Interface (LPI) Version 4.1*, 1997.

[97] O. Levia. *Programming System Architectures with Java.* IEEE Computer, August 1999.

[98] S. Lewis. *The art and science of Smalltalk*. Prentice-Hall, London, 1995.

[99] S. Liao, S. Tjiang, R. Gupta. *An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment*. Proc. Design Automation Conference (DAC), 1997.

[100] B. Lin, S. Vercauteren, H. De Man. *Embedded Architecture Co-Synthesis and System Integration*. Proc. International Workshop on Hardware/Software Codesign (CODES), 1996.

[101] S. Maginot, W. Nebel, W. Putzke-Röming, M. Radetzki. *Final Objective VHDL language definition*. REQUEST Deliverable 2.1.A (public), 1997. Available from http://eis.informatik.uni-oldenburg.de/research/request.html

[102] S. März. *High-Level Synthesis*. In: P. Michel, U. Lauther, P. Duzy (eds): The Synthesis Approach to Digital System Design. Kluwer Academic Publishers, 1992.

[103] S. Matsuoka, A. Yonezawa. *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*. In: G. Agha,

P. Wegner, A. Yonezawa (eds), Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993.

[104] M. Meixner, J. Becker, Th. Hollstein, M. Glesner. *Object-oriented Specification Approach for Synthesis of Hardware-/Software Systems*. Proc. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, 1999.

[105] O. Mencer, M. Morf, M. J. Flynn. *PAM-Blox: High Performance FPGA Design for Adaptive Computing*. Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), 1998.

[106] B. Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1992.

[107] L. H. Miller. *Advanced programming: design and structure using PASCAL*. Addison-Wesley, Reading, Mass., 1986.

[108] M. T. Mills. *Proposed Object Oriented Programming Enhancements to the Very High Speed Integrated Circuits Hardware Description Language (VHDL)*. Report 45433-7331, Wright Laboratory / Wright-Patterson Air Force Base, Ohio, 1993.

[109] D. Morris, G. Evans, P. Green, C. Theaker. *Object-oriented Computer Systems Engineering*. Springer, London, 1996.

[110] W. Nebel, G. Schumacher. *Object-Oriented Hardware Modelling—Where to apply and what are the objects?* Proc. EURO-DAC with EURO-VHDL, 1996.

[111] W. Nebel, W. Putzke-Röming, M. Radetzki. *Das OMI-Projekt REQUEST* (German). Proc. 3. GI/ITG/GMM Workshop Hardwarebeschreibungssprachen und Modellierungsparadigmen, 1997.

[112] O. Nierstrasz. *Composing Active Objects.* In: G. Agha, P. Wegner, A. Yonezawa (Eds.): Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993.

[113] F. Oppenheimer, G. Schumacher, W. Nebel. *Modellierung und Simulation eines Portalkrans mit der OO-COSIM Methode* (German). To appear, Proc. AES 2000, Karlsruhe.

[114] I. Page. *Constructing Hardware-Software Systems from a Single Description*. Journal of VLSI Signal Processing, 12(1), 1996, pp. 87-107.

[115] C. Passerone et al. *Modeling Reactive Systems in Java*. ACM Transactions on Design Automation of Electronic Systems, vol. 3, no. 4, 1998, pp. 515-523.

[116] A. Pawlak, W. Wrona. *Modern Object-Oriented Programming Language as a HDL*. Proc. Conf. on Computer Hardware Description

Languages (CHDL), 1987.

[117] J. S. Poulin. *Measuring Software Reuse: Principles, Practices, and Economic Models*. Addison-Wesley, 1997.

[118] W. Putzke-Röming, M. Radetzki, W. Nebel. *Objective VHDL: Hardware Reuse by Means of Object-Orientation*. Proc. 1st Workshop on Reuse Techniques in VLSI Design, 1997.

[119] W. Putzke-Röming, M. Radetzki, W. Nebel. *A Flexible Message Passing Mechanism for Objective VHDL*. Proc. Design, Automation and Test in Europe (DATE), 1998.

[120] W. Putzke-Röming, M. Radetzki, W. Nebel. *Modeling Communication with Objective VHDL*. Proc. VHDL Int'l User Forum (VIUF, Spring Conference), 1998.

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

[122] M. Radetzki, W. Putzke-Röming, W. Nebel. *OO-VHDL: What Is It, and Why Do We Need It?* Proc. Asia-Pacific Conference on Hardware Description Languages (APCHDL), 1997.

[123] M. Radetzki, W. Putzke-Röming, W. Nebel. Objective VHDL: The Object-Oriented Approach to Hardware Reuse. In: J.-Y. Roger, B. Stanford-Smith, P. T. Kidd (eds.): *Advances in Information Technologies: The Business Challenge*. IOS Press, 1998.

[124] M. Radetzki, W. Putzke-Röming, W. Nebel. *Übersetzung von Objektorientiertem VHDL nach Standard VHDL* (German). Proc. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, 1998.

[125] M. Radetzki, W. Putzke-Röming, W. Nebel. *Objective VHDL: Tools and Applications*. Proc. Forum on Design Languages (FDL), 1998.

[126] M. Radetzki, W. Putzke-Röming, W. Nebel. *A Unified Approach to Object-Oriented VHDL*. Journal of Information Science and Engineering 14 (1998), pp. 523-545.

[127] M. Radetzki, W. Putzke-Röming, W. Nebel. *First Release Objective VHDL*. ESPRIT project 20616 (REQUEST) deliverable 2.1.E, 1998.

[128] M. Radetzki, A. Stammermann, W. Putzke-Röming, W. Nebel. *Data Type Analysis for Hardware Synthesis from Object-Oriented Models*. Proc. Design, Automation and Test in Europe (DATE), 1999.

[129] M. Radetzki, W. Nebel. *Synthesis of Hardware Structures from Object-Oriented Models*. Proc. 8th Int'l Symposium on Integrated Cir-

cuits, Devices and Systems (ISIC), 1999.
[130] M. Radetzki. *Overview of Objective VHDL Language Features*. Proc. Forum on Design Languages (FDL), 1999.
[131] M. Radetzki, W. Nebel. *Synthesizing Hardware from Object-Oriented Descriptions*. Proc. 2nd Forum on Design Languages (FDL), 1999.
[132] M. Radetzki. *Preliminary Specification of Objective VHDL Synthesizability Rules*. Esprit project 28889 (SQUASH) deliverable D2.5.A, 1999.
[133] C. R. Ramesh. *Object Orienting VHDL for Component Modeling*. VHDL International User Forum (VIUF, fall conference), 1994.
[134] F. Rammig. *Systematischer Entwurf digitaler Systeme* (German). Teubner, Stuttgart, 1989.
[135] RASSP Taxonomy Working Group. *VHDL Modeling Terminology and Taxonomy*, Revision 2.4. RASSP document, http://rassp.scra.org, 1998.
[136] J. Riesco Prieto et al. *Report on the final evaluation of the REQUEST methodology and toolsets*. Esprit Project 20616 (REQUEST) deliverable D3.3.A, 1998.
[137] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
[138] J. Rumbaugh, I. Jacobson, G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1998.
[139] G. Salamuniccar. *A Proposal for Data Modeling Extension to VHDL Using an Object-Oriented Approach*. Proc. WDTA, 1998.
[140] S. Sarkar. *An Object Oriented Approach to Digital Circuit Synthesis*. Dissertation, Indian Institute of Technology, 1995.
[141] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, I. Bolsens. *A Programming Environment for the Design of Complex High Speed ASICs*. Proc. Design Automation Conference (DAC), 1998.
[142] G. Schumacher, W. Nebel. *Inheritance Concept for Signals in Object-Oriented Extensions to VHDL*. Proc. EURO-DAC with EURO-VHDL, 1995.
[143] G. Schumacher, W. Nebel, W. Putzke, M. Wilmes. *Applying Object-Oriented Techniques to Hardware Modelling—A Case Study.* Proc. VHDL User Forum Europe (VUFE), 1996.
[144] G. Schumacher. W. Nebel. *Object-Oriented Modelling of Parallel Hardware Systems*. Proc. Design, Automation and Test in Europe (DATE), 1998.
[145] G. Schumacher. *Object-Oriented Hardware Specification and Design*

with a Language Extension to VHDL. Ph.D. dissertation, Oldenburg University, 1999.

[146] B. Selic, G. Gullekson. P. T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, New York, 1994.

[147] L. Sèmèria, G. De Micheli. *SpC: Synthesis of Pointers in C, Application of Pointer Analysis to the Behavioral Synthesis from C*. Proc. International Conference on Computer Aided Design (ICCAD), 1998.

[148] Semiconductors Industry Association. *The National Technology Roadmap for Semiconductors*. 1997.

[149] J. Siegel, H. Eichele. *Hardwareentwicklung mit ASIC* (German). Hüthig Buch, Heidelberg, 1990.

[150] A. Stammermann. *Datentypanalyse objektorientierter Hardwarebeschreibungen* (German). Diploma thesis, Oldenburg University, 1998.

[151] B. Stroustrup. *The C++ programming language*. 3rd ed., 9th print. Addison-Wesley, Reading, Mass., 1999.

[152] Synopsys, Inc. *Behavioral Compiler User Guide*, version 1998_08, 1998.

[153] Synopsys, Inc. *DesignWare User Guide*, version 1998_08, 1998.

[154] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, New York, 1998.

[155] A. Takeuchi. *Object Oriented Description Environment for Computer Hardware*. Proc. Computer Hardware Description Languages and their Applications (CHDL), 1981.

[156] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[157] D. E. Thomas, P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[158] K. Van Rompaey, D. Verkest. I. Bolsens. H. De Man. *CoWare—A design environment for heterogeneous hardware/software systems*. Proc. EURO-DAC, 1996.

[159] S. Vercauteren, B. Lin, H. De Man. *Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications*. Proc. Design Automation Conference (DAC), 1996.

[160] D. Verkest et al. *Matisse: A system-on-chip design methodology emphasizing dynamic memory management*. Proc. IEEE CS Workshop on VLSI (IWV), 1998.

[161] P. Wegner. *Dimensions of Object-Based Language Design*. ACM SIGPLAN Notices, vol. 22, no. 12, Proc. OOPSLA'87, 1987.

[162] W. H. Wolf. *How to Build a Hardware Description and Measurement System on an Object-Oriented Programming Language*. IEEE Trans-

actions on Computer-Aided Design, vol. 8, no. 3, 1989.

[163] VSI Alliance. *VSI System Level Design Model Taxonomy, Version 1.0*. VSI Document, http://www.vsi.org, 1998.

[164] J. Willis. S. Bailey, R. Newshutz. *A Proposal for Minimally Extending VHDL to Achieve Data Encapsulation, Late Binding, and Multiple Inheritance*. Proc. VHDL International Users' Forum (VIUF), 1994.

[165] J. Willis, S. Bailey, C. Swart. *Shared Variable Language Change Specification* (PAR 1076A), Version 5.7, 1996.

[166] M. Wilmes. *Hardware-Spezifikationn mit objektorientierten Sprachererweiterungen zu VHDL* (German). Diploma thesis, Oldenburg University, 1995.

[167] J. S. Young, A. R. Newton. *Embedding programs in the Java language in the synchronous model of computation through the process of successive, formal refinement*. Proc. International Conference on Computer-Aided Design (ICCAD), 1997.

[168] J. S. Young, J. MacDonald, M. Shilman, P. H. Tabbara, A. R. Newton. *Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement*. Proc. Design Automation Conference (DAC), 1998.

[169] J. S. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, A. R. Newton. *The JavaTime Approach to Mixed Hardware-Software System Design*. In: A. A. Jerraya, J. Mermet: System-Level Synthesis. NATO Science Series E, vol. 357, Kluwer Academic Publishers, 1999.

[170] ZAM-Anwendungszentrum Nürnberg. *Ordnung mit Objektklassen im Design-Prozeß* (German). Design, Fertigung & Test, Issue 9/96, 1996.

[171] R. Zippelius, K. D. Müller-Glaser. *An Object-oriented Extension of VHDL*. Proc. VHDL Forum for CAD in Europe (VFE, Spring Conference), 1992.

Curriculum Vitae

04/00 to date	Sican GmbH, Hannover, Design Engineer
08/96 – 03/00	OFFIS (Oldenburg Research and Development Institute for Informatic Tools and Systems), Research Scientist
10/91 – 07/96	Oldenburg University, Diploma in Informatics
12. 11. 1971	Born in Hildesheim