MASTER

Part of construction of a TL to VHDL compiler

van Itegem, Jean-Paul R.A.

Award date:
1993

Link to publication
part of

Construction of a TL to VHDL compiler

Ing. J-P R.A. v. Itegem

By: Ing. J-P R.A. v. Itegem (320359)
student at the Eindhoven University of Technology, department of Information Engineering.

Supervisors: Ir. L.P.M. Benders and Prof. Ir. M.P.J. Stevens
Eindhoven University of Technology, department of Electrical Engineering, section of Digital Information Systems

Eindhoven, August '93

The Eindhoven University of Technology does not accept any responsibility for the contents of reports made by students.
ABSTRACT

The large complexity of contemporary digital systems makes design automation tools indispensable. These tools automate the system synthesis, allowing the designer to concentrate on the system's functionality. However these tools require a system description in a language they understand, therefore the specification language TL has been designed. Embedded systems described with TL can be automatically analyzed, and based on the results the system can be split in a software and a hardware part.

For synthesis purposes, the TL description has to be translated to IEEE VHDL. As the basis for such a compiler an implementation of an environment tree (a data structure containing all declared items) and access routines obeying the scope and visibility rules of TL, has been made. On this basis a compiler has been built to map three TL language constructs (events, mutual exclusion and communication) to VHDL.

The TL event constructs, which can be used to synchronise concurrent processes, could be implemented using one signal controlled by a resolution function.

For the mutual exclusion constructs a data bus is used to model the mutual exclusion object. Because the data bus model offered by VHDL proved to be inadequate in a hierarchical design, a data bus emulation scheme, using a special resolution function in combination with an extension to the type of the data bus, has been designed. The mutual exclusion algorithm is implemented by an arbiter. There are three other algorithms recognized. One, a set of arbiters hierarchically organized to adapt to the structural hierarchy of the design. Two, an algorithm using a token-ring, which fit equally well in the hierarchy. Three, a mutual exclusion algorithm with priorities, either implemented using one central arbiter or a series of arbiters, one for each priority level.

The TL communication construct supports communication between one sender and a set of explicitly specified receivers. Several senders can be made to share one channel. Two protocols are used to implement the synchronization of the senders and receivers. The shared channels are protected by including mutual exclusion in the protocol.

The type of the channel implicitly depends on the types of the messages sent over it. The declarations of these types are moved to a special package at the top of the VHDL file so that they can be made visible to the declarations implementing the channel.

Communication can benefit from buffering. The place and size of the used buffers determines the gain. The best place depends on the behaviour of the sender and the receiver, and the utilisation of the channel. The conclusion of comparing two buffer implementations, a ring-buffer and a large shift-register, is that the shift-register only has advantages as a small buffer in hardware. For large buffers and in software the ring-buffer is better.
This report is written by Jean-paul v. Itegem, student Information Engineering (IT) at the Eindhoven University of Technology (TUE), as a part of my graduation assignment. The assignment was given by and carried out at the TUE, department Electrical Engineering, section Digital Information Systems.

One group of this section develops methods to map hierarchical descriptions of digital systems to a processor architecture that is optimal for the given problem. For this the designer uses, after determining the system specifications, an algorithmic language to describe the design. An extended version of IEEE VHDL [IEEE] has been designed as the algorithmic language. The extensions to IEEE VHDL include abstract synchronisation and communication primitives.

The extended VHDL language is called Task Level VHDL (TLVHDL) or simply TL [Ben1-4]. For analysis of the system the description is translated to ExSpect (coloured Petri nets) [Hoo]. For synthesis purposes the system description is translated to standard IEEE VHDL. Based on the results of the analysis and design knowledge, the system can be split into a software and a hardware part. It is expected that in the future tools will become available to further synthesize the hardware part from the VHDL description.

My assignment was to continue the work of Ir. M.V. Boersma who started the construction of a compiler (using Lex and Yacc) to translate TL to VHDL. The assignment incorporates designing implementation for parts of the TL language in VHDL and constructing a compiler that transforms TL to VHDL. In constructing the compiler it was assumed that the description has passed the front end of the compiler and thus is correct. A large part of the front end has been constructed by Ir. O. Pigmans [Pig].

Because the tools Lex and Yacc only support a small part of the compiler construction, it was decided that a new compiler had to be written using the GMD-toolbox for compiler construction. This toolbox, made at the University of Karlsruhe (Germany) [Gro], supports the full trajectory of compiler construction. The software was available on an Apollo system in a Unix environment. The programming language used is C [Ker]. To test possible VHDL implementations the V-system/Windows VHDL simulator of Model Technology Incorporated [MTI] was available on a PC/AT with a MS-DOS environment.
I want to thank the following people for their help and support in executing my assignment:
- Ir. L.P.M. Benders, head of our project group and my supervisor
- Prof. Ir. M.P.J. Stevens, head of the section Digital Information Systems
- Ir. M.V. Boersma, my predecessor
- Ir. M. Hoorn, designer of TL to ExSpect compiler
- Ir. O. Pigmans, designer of front end for TL compiler
- H.W.J. v. Gaalen, secretary
- Ing. A. Wijfels, system manager
- Ir. W.J. Withagen, GMD-toolbox expert
- Ing. J.P.M. v. Itegem, a great help
Further I want to thank every body else that supported me during this last year of my studies.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>TL TO VHDL COMPILER</td>
<td>55</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>Implementing an environment tree</td>
<td>56</td>
</tr>
<tr>
<td>4.3</td>
<td>Compiler for events</td>
<td>59</td>
</tr>
<tr>
<td>4.4</td>
<td>Compiler for mutual exclusion</td>
<td>60</td>
</tr>
<tr>
<td>4.5</td>
<td>Compiler for communication</td>
<td>61</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Setting channel types</td>
<td>62</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Gathering implicit subchannels</td>
<td>63</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Setting the channel direction</td>
<td>63</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Computing mutual exclusion numbers for communication</td>
<td>64</td>
</tr>
<tr>
<td>4.5.5</td>
<td>Computing the required HSM-sender processes</td>
<td>64</td>
</tr>
<tr>
<td>4.5.6</td>
<td>Computing other communication context information</td>
<td>65</td>
</tr>
<tr>
<td>4.6</td>
<td>Gathering intermediate types for 'special_types'</td>
<td>66</td>
</tr>
<tr>
<td>5</td>
<td>EXTENSIONS TO THE COMPILER</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>Extensions to the event implementation</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Extensions to the mutual exclusion implementation</td>
<td>67</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Hierarchical arbiter</td>
<td>68</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Token-ring mutual exclusion algorithm</td>
<td>69</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Mutual exclusion with priorities</td>
<td>70</td>
</tr>
<tr>
<td>5.3</td>
<td>Extensions to the communication implementation</td>
<td>72</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Optimizing communication types</td>
<td>72</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Buffers for communication</td>
<td>72</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Protocols with receiver initiative</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>CONCLUSIONS</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>Appendix A: 'TL_TYPES' library file</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>Appendix B: An example</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>INDEX</td>
<td>105</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

- "Bleep Bleep .. What are your wishes master .. Bleep".
- "Eeh..., I would like to have a system that shows the time and that I can program to wake me in the morning with gentle music".
- "Bleep Bleep .. Your wish is my command. I will immediately start designing such a system. It will be ready tomorrow at ten thirteen .. Bleep".

The above dialogue might be recorded over several decades when designing systems is fully automated and the designing system is equipped with a speech interface. Alas at present this is science fiction, but that does not mean that we cannot try to make steps in that direction to close the tremendous gap. On the other hand we might make a more intelligent system that answers: "Those kinds of systems can be found in the museum ...".

This report describes a small part of a step in the direction of automating the design of embedded systems. The project (the step) aims at developing methods to map hierarchical descriptions of digital systems to a processor architecture that is optimal for the given problem. An extended version of IEEE VHDL is used for the hierarchical description of digital systems. This new language, called Task Level VHDL (TLVHDL or TL for short) [Benl-4], is more abstract than standard VHDL and thus allows the designer to describe the system on a higher level (closer to the natural language).

For synthesis purposes the TL description has to be translated to an equivalent IEEE VHDL description. This report describes the translation of the event, mutual exclusion and communication constructs of TL to VHDL. It also describes globally the approach of the compiler to execute the translations. The compiler was build using the GMD-toolbox [Gro]. A part of the implementation of the event and mutual exclusion constructs and to a lesser extend also communication, is based on the work of Ir. M.V. Boersma who started the construction of a TL to VHDL compiler using the tools Lex and Yacc [Boel].

This report will facilitate updating VHDL implementations, finishing the compiler and maintaining the constructed part of the compiler. Next to giving the reasons why an implementation was chosen, it tells why rejected ideas were not satisfactory. Chapter 2 contains brief introductions to respectively VHDL, TL, compiler construction and the use of the GMD-toolbox.

Chapter 3 describes the syntax, semantics and VHDL implementation of events, mutual exclusion and communication constructs. The next chapter describes the structure of the compiler that was build to translate these three TL constructs and an implementation of an environment tree. Chapter 5 contains ideas for alternative implementations of the three TL constructs, extensions to the TL language and optimizations for the compiler.

Appendix A contains an example that demonstrates some of the capabilities of the current TL to VHDL compiler.
Some footnotes in this report are intended for insiders on VHDL, the VHDL introduction contained in this report certainly is insufficient to grasp these remarks. Likewise this report contains several texts listing the VHDL code that implements some TL language construct. These codes are merely included for completeness, they are not intended to clarify the concept of the implementation.
This chapter contains brief introductions to respectively VHDL, TL and compiler construction with the GMD-toolbox. These introductions are purely background information and should be sufficient to understand the rest of this report. Parts of the first two sections are borrowed from Ir M. Boersma's report [Boel].

2.1 Introduction to VHDL

The hardware description language VHDL, VHSIC (Very High Speed Integrated Circuits) Hardware Description Language [IEEE] [Lip] [Ash], can be used to model and simulate digital systems. With VHDL both functional and structural descriptions can be combined into a hierarchical design.

2.1.1 Describing a system in VHDL

A system is represented in VHDL by an entity, which consists of an entity declaration and an architecture body. The entity declaration describes the interface of the system, i.e. it describes its inputs and outputs (see figure 2.1). The architecture body describes the behaviour of the entity or in other words it describes how the outputs are related to the inputs.

\[
\begin{align*}
    a & \rightarrow \text{1} \rightarrow y \\
    b & \rightarrow y
\end{align*}
\]

*Fig. 2.1: Interface of an EXOR entity.*

In VHDL the behaviour can be described in three ways. First of all the behaviour of an entity can be described by describing its structure. Such a structural description consists of components, which are connected to each other and to the inputs and outputs of the entity. Connections can be established via signals which are an abstraction of wires. The components are instantiations of other entities. As an example a structure of an EXOR is shown in figure 2.2.

The second way to describe the behaviour of an entity is using one or more functional descriptions, for example \( y = \overline{a} \cdot b + a \cdot \overline{b} \) for an EXOR. Each functional description is represented by a process. The function of a process is described using programming constructs like those in the procedural languages Pascal or C (e.g. statements and procedure calls). In VHDL these processes execute simultaneously. The processes can interact with each other and with the in- and outputs using signals.

The third and last possibility to describe the behaviour of an entity is mixing structural and functional descriptions. Again signals can be used to make the processes and
instantiated components interact with each other.

The next example shows the VHDL code for the above named descriptions for the EXOR gate. Text 2.1 lists the entity declaration for the EXOR (see also figure 2.1). Notice that VHDL refers to the interface of an entity with the name 'port'.


defined components interact with each other.

The next example shows the VHDL code for the above named descriptions for the EXOR gate. Text 2.1 lists the entity declaration for the EXOR (see also figure 2.1). Notice that VHDL refers to the interface of an entity with the name 'port'.
ARCHITECTURE structure_exor OF entity_exor IS  
COMPONENT entity_inverter  
  PORT (a : IN bit;  
    na : OUT bit);  
END COMPONENT;  
COMPONENT entity_and  
  PORT (a, b : IN bit;  
    y : OUT bit);  
END COMPONENT;  
COMPONENT entity_or  
  PORT (a, b : IN bit;  
    y : OUT bit);  
END COMPONENT;  
SIGNAL na, nb: bit;  
SIGNAL s1, s2: bit;  
BEGIN  
  -- positional interface association  
  inv_a: entity_inverter PORT MAP (a, na);  
  -- named interface association  
  inv_b: entity_inverter PORT MAP (a=> b,  
    na=> nb);  
  and1: entity_and PORT MAP (na, b, s1);  
  and2: entity_and PORT MAP (a, nb, s2);  
  or1: entity_or PORT MAP (s1, s2, y);  
END;

Text 2.2: Structural description of the behaviour of an EXOR.

ARCHITECTURE function_exor OF entity_exor IS  
BEGIN  
  exor_proc: PROCESS  
    BEGIN  
      y <= ((NOT a) AND b) OR (a AND (NOT b));  
    END PROCESS;  
END;

Text 2.3: Functional description of an EXOR.

ARCHITECTURE mix_exor OF entity_exor IS  
COMPONENT entity_or  
  PORT (a, b : IN bit;  
    y : OUT bit);  
END COMPONENT;  
SIGNAL s1, s2: bit;  
BEGIN  
  exor_part1: PROCESS  
    VARIABLE na: bit;  
    BEGIN  
      na:= NOT a;  
      s1<= na NAND b;  
    END PROCESS;  
  exor_part2: PROCESS  
    VARIABLE nb: bit;  
    BEGIN  
      nb:= NOT b;  
      s2<= a NAND nb;  
    END PROCESS;  
  or1: entity_or PORT MAP (s1, s2, y);  
END;

Text 2.4: Mixed functional and structural description of an EXOR.

Therefore the entity in which a object is declared local is called the highest hierarchical level on which the object is used. The term 'local process' refers to a process declared within the discussed entity. A 'local communication' is a communication between local processes.
2.1.2 Using declarations and packages

In the above example of the EXOR entity all signals had the type 'bit'. This is a standard type and it is intended for signals that model wires in a digital system. Next to this simple type VHDL offers a rich variety of data types. Similar to the procedural programming languages C and Pascal, VHDL allows the designer even to construct complex data types with record and array declarations. These complex data types can be declared in an entity declaration or in the declaration part of a process or an architecture body. Except types, VHDL also knows subtypes, which are a constrained versions of a types or other subtypes. Other useful declarations in VHDL are those of subprograms, i.e. functions and procedures.

VHDL, like the Pascal, enforces strong typing. Strong typing means that two type declarations which are exactly the same, i.e. the same identifier (declared in different places) and the same type definition, are NOT considered the same type. Furthermore, VHDL demands that signals that are connected to each other (e.g. in an interface association) have the same type. Likewise the right side of an assignment (both signal and variable assignment) must have the same type as the object to which is assigned. Due to strong typing this means that the same type declaration has to be used for the declaration of both items, be it a connection or an assignment. On the other hand, VHDL considers all subtypes of one type to be of the same type.

By using packages declarations of data types or subprograms can be used in more than one entity. Like entities, packages consist of a declaration part and a body part. The collective noun for package and entity declarations is primary unit. Package bodies and architectures (entity bodies) are indicated by secondary unit. The term design unit refers to both primary and secondary units. The declarations in the package declaration part can be made visible in other entities or packages with a use clause. Usually the package declaration contains, among other things, the declaration parts of subprograms, while the corresponding package body, whose contents cannot be made visible elsewhere, contains the body parts of those subprograms.

Text 2.5 lists an example VHDL file to clarify the use of packages and declarations. The use clause in the example, like most use clauses, consists of three parts. The identifier 'WORK' is a standard VHDL identifier that always refers current working directory. This identifier indicates that the package declaration to be made visible is located there (maybe even in the same file). The second identifier, 'test_package' in the example, identifies the intended packages and the third identifier is the identifier of the declaration to be made visible. Here the reserved word 'ALL' indicates that all declarations from the package declaration have to be made visible.
-- comment starts with two dashes and ends at the end of the line

PACKAKE test_package IS
  CONSTANT delay: Time := 1 NS;  -- constant type 'Time' of 1 nano second
  TYPE test_record IS  -- a record containing two fields
    RECORD
      field1: BOOLEAN;
      field2: INTEGER;
    END RECORD;
  -- an array of 6 records
  TYPE test_array IS ARRAY (0 TO 5) OF test_record;
  -- a procedure declaration
  PROCEDURE test_procedure (VARIALE test_var: INOUT test_record);
END;

PACKAKE BODY test_package IS
  PROCEDURE test_procedure (VARIALE test_var: INOUT test_record) IS
    VARIALE local_var: test_record;
    BEGIN
      local_var := (FALSE, 0);  -- an initialization of the local variable
      test_var.field2 := local_var.field2;
    END;
END pak;

USE WORK.test_package.ALL;  -- a use clause

ENTITY test_entity IS  -- no interface
END;

ARCHITECTURE test_architecture OF test_entity IS
  -- subtypes can be used to limit the range of types
  SUBTYPE limited_type IS INTEGER RANGE (0 TO 15);
  SIGNAL test_wire: test_array;  -- a 'wire' of 6 records
  BEGIN
    test_process: PROCESS
      VARIALE test_variable: test_record;
      BEGIN
        test_procedure (test_variable);
        test_wire (2) <= test_variable;
      END PROCESS;
  END;

Text 2.5: Example VHDL file with a package and a use clause.

2.1.3 Scope and visibility rules

In the previous subsections it was shown that declarations (e.g. of types) are allowed in several places. When an identifier is used in statements it always refers to one such declaration. Which declaration eventually is used (seen) is governed by the scope and visibility rules. VHDL formulates scope and visibility rules using the notion of

1 Actually also the overloading rules have to be applied, but those are not considered yet and therefore are outside the scope of this report.
declarative region. A declarative region consist of one or two declaration parts together with the corresponding statement part (see the example in text 2.6). Examples of declarative regions are:

- a primary unit with the corresponding secondary unit
- a subprogram declaration, together with the corresponding subprogram body
- a record type declaration
- a process-statement

The scope of a declaration extends from the beginning of the declaration to the end of the declarative region.

---

Text 2.6: Example showing definitions for the scope and visibility rules.

Two declarative regions are said to be nested if one declarative region is surrounded by the other (e.g. the processes in the text). If an identifier is declared in each of two nested declarative regions then the declaration in the inner declarative region is said to hide the declaration of the outer declarative region. Identifier declarations are said to be directly visible within their scope unless they are hidden. So in the example, the first 'sig1' declaration is directly visible from its declaration till the end of the architecture body, except where it is hidden. This hidden part is the scope of the 'sig1' variable
declaration in the first process, which extends from the 'sig1' variable declaration till the end of the process.

Declarations in a package declaration can be made indirectly visible in another design unit with a use clause. A declaration from a package can only be made indirectly visible if the same identifier is not already directly visible (thus not declared in any of the surrounding declarative regions). This implies that a directly visible declaration can never be hidden by a indirectly visible declaration. Thus in the example of text 2.6, 'sig3' from package 'pak2' does not become visible because of the 'sig3' signal declaration in the architecture body.

Note that a use clause of a package named A only makes indirectly visible the declarations that occur in the package declaration of package A. It does NOT make visible the declarations that were made indirectly visible by a use clause for the package declaration of package A! So in the example the use clause for 'pak2' does not make visible the declarations from 'pak1' in spite of the fact that the declarations from 'pak1' were made visible in 'pak2'.

Also note that as soon as an identifier is made indirectly visible more than once it is an error to use that identifier. For this it is not significant whether or not it is exactly the same declaration that was made visible, nor does it matter whether or not the first use clause is in the declarative region surrounding the declarative region where the second use clause is used. The latter means that identifiers made indirectly visible through a use clause can never hide each other.

2.1.4 Using resolution functions

Most digital systems have one or more wires or data busses that have more than one driver (in VHDL each process assigning a value to a data bus contains a driver for that data bus). A simple example of such wires is the wired-AND. In VHDL such a wire can be modelled, but to be able to simulate such a wire, a so-called resolution function is required. This VHDL feature is an essential part of the VHDL code used to translate TL constructs, therefore this feature is shortly explained here.

The problem the VHDL simulator faces is that it has to figure out the value for every signal (wire). This is easy when there is only one driver writing a value to the signal, however it is a problem when there is more than one driver for the signal. In VHDL this complex problem is solved by a resolution function. A resolution function is a function that has as input parameters all the values (in an array) assigned to the signal by the drivers. With this information the function calculates what the eventual value for the signal should be and returns it. A resolution function is written by the designer modelling the digital system. So the designer is free to make a resolution function that models a wired-OR, a wired-AND, colour mixing or even more exotic things.

The resolution function is called by the simulator and the simulator also ensures that the returned value is the value detected by any process reading the signal (even if the process just assigned another value to the same signal). Before the simulator can call a
resolution function to determine a value for a signal it has to know which function it has to call. This is done by incorporating the resolution function, via a *subtype declaration*, in the signal's type. Text 2.7 lists an example with a wired-OR resolution function.

---

**ARCHITECTURE** beh **OF** system **IS**

**TYPE** bit_array **IS** ARRAY (**NATURAL** RANGE<>) **OF** BIT; -- declare an unconstrained array

**FUNCTION** resolve_bit **(driver: IN** bit_array **) RETURN** BIT;

**SUBTYPE** wired_or **IS** resolve_bit BIT; -- associate the function with the type

**FUNCTION** resolve_bit **(driver: IN** bit_array **) RETURN** BIT **IS**

BEGIN

-- loop variables are automatically declared, RANGE is an attribute of driver

FOR i **IN** driver**RANGE** LOOP

IF driver(i) = '1' **THEN**

RETURN '1';

END IF;

END LOOP;

RETURN '0';

END;

**SIGNAL** wire: wired_or;

BEGIN

proc_a: **PROCESS**

BEGIN

wire <= '0';

WAIT FOR 1 NS;

wire <= '1';

WAIT;

END PROCESS;

-- unconditional wait suspends process

proc_b: **PROCESS**

BEGIN

wire <= '0';

WAIT;

END PROCESS;

END;

---

Text 2.7: Example using a resolution function.

### 2.1.5 Modelling data busses in VHDL

As stated before, digital systems usually have at least one data bus with more than one driver. For such a case it has to be ensured that there is not more than one driver writing a value to the bus at any time. That implies that all bus drivers must have the possibility to be *disconnected* from the bus. In VHDL the keyword *'NULL'* was introduced specially for modelling disconnecting drivers (i.e. processes) from busses. Alas this scheme proves to be inadequate in a design with hierarchy.

If there is more than one potential driver for a signal then a resolution function is compulsory. So there also has to be a resolution function for a data bus for which there is never more than one driver connected. However it turns out that there is a problem
when all drivers are disconnected in a design with a structural hierarchy. In such a design the simulator acts as depicted in figure 2.3 (at least the used simulator does). The INOUT port in the figure is the bus interface of the entity. This has to be an INOUT port because the bus can be read and written. The problem is caused by the fact that the simulator uses the resolution function for every entity (hierarchical level) in the design. So in the example the resolution function is called to resolve the values written by the two drivers in the lower entity and the returned value is made visible to the resolution function of the higher level via the OUT part of the port.

Fig. 2.3: Resolving signals over a structural hierarchy.

When both drivers in the entity are disconnected from the bus, the resolution function is called with an array of zero elements. This situation can be detected by the resolution function and it should react on that by disconnecting the entity as a whole from the bus. One would expect that using the 'NULL' keyword as a parameter of the return-statement would do the trick. Alas it turns out that VHDL does not allow this!! This is because the 'NULL' keyword can only be used in signal assignments and a return-statement is not a signal assignment.

Consequently this VHDL feature proves to be quite useless in a design with a structural hierarchy. The described problem has been solved by making an own implementation to disconnect drivers from a bus. This is further explained in section 3.3.2.1, where the implementation of mutual exclusion objects is discussed.

2.2 Introduction to TL

In VHDL signals play a central role. A designer using VHDL to model an embedded system (a combination of hardware and software dedicated to perform a specific job) will spend a lot of time on correctly connecting signals and implementing protocols for/ on those signals. Such a protocol has to make sure that data cannot get lost, that the

---

1 The problem would be solved if the simulator would not call the resolution function for each entity but only for the whole design.
data reception is acknowledged, that processes are synchronized, that mutual exclusion protection is implemented for a commonly used object, etc.

This method has two major disadvantages. First of all the designer spends precious time on specifying HOW a task is done while instead that time should be spent on specifying WHAT the task has to do. Secondly the result of all the designers effort is a piece of hard to comprehend VHDL text. The latter is a consequence of the fact that VHDL shows what the designer has implemented i.e. it shows HOW. An indirect disadvantage of this problem is that it is virtually impossible to automatically analyze and optimize VHDL files.

The language Task Level VHDL is designed to solve the above problems building on the good basics of VHDL. The constructs offered by TL take care of synchronization, mutual exclusion, communication, process sharing, interrupt handling and process control flow. The good things kept from VHDL incorporate hierarchy (entities), the procedural language, packages, the complex data types, the scope and visibility rules and strong typing. In fact TL removes signals from VHDL and replaces them by the previously named constructs. Those constructs specify what is done while hiding how it is done. In doing so automatic analyzation and optimization of the system specifications and thus system synthesis becomes feasible. The latter is not an accidental spin off of the designed language, but was the primary reason for defining this new language.

To ease the analyzation and optimization, statements can be grouped together in a so-called leaf-task. Since for communication statements (and others) the name of the leaf-task in which they occur is relevant, they cannot occur outside a leaf-task. Text 2.8 shows a small TL example. Note that TL allows only one TL specific statement (statements known by TL and not known by VHDL) in each leaf-task.

Another major difference between TL and VHDL is that the concept time is almost completely removed. All TL specific statements are said to take one time unit to execute. The total time it takes to finish a TL statement is that time unit plus the time the statement is held up by one or more other processes. The only way this 'time' can be used in a TL specification is by specifying a time-out condition for communication, synchronization or other TL actions.
2.3 Introduction to compiler construction

A significant part of this report is dedicated to explaining what actions the compiler takes to translate TL to VHDL. For this explanation some knowledge of compiler notions is required, therefore these notions are briefly introduced here. For a more extensive story on this subject one can read the report of O. Pigmans [Pig], lecture notes for the course Compilers [DMC] or the book by T. Pittman [Pit].

2.3.1 Structure of a compiler

Virtually every compiler makes use of four building blocks (see fig. 2.4), namely a (lexical) scanner, a parser, a context dependent analyzer and a code generator. The assignment for the scanner is to scan the source file and to transform all recognized 'words' to tokens. A word is in this context a group of characters belonging together and a token is a representation of such a word. For example the word 'IF' can be
transformed to an if_token, the word '/' to an operator_token, the word 'vakantie_toeslag' to an identifier_token, etc.

The tokens recognized by the scanner are passed on to the parser. The parser uses the tokens and the order of the tokens to build a parser tree. The parser tree is built according to the (concrete) syntax of the language and it is a one to one mapping of the structure of the source code. As an example a basic if-then-else language construction is represented in the parser tree by one node having three children. The first child contains the (sub)tree representing the condition of the if-statement. The second child contains the statements to be executed when the condition evaluates to true (the then_statements) and the third child contains the else_statements. How this looks like for an if-statement is illustrated by figure 2.5.

IF a>b THEN max:=a ELSE max:=b;

In the example some important information is not included in the parser tree. As one can see all identifiers are represented by an identifier node. This is sufficient to represent the structure of the source file, but the compiler also needs to know what identifier is represented by each identifier node. This kind of information is inserted in the parser tree by extending tree nodes with attributes. Another attribute commonly
added by the parser is the position in the source text of the language constructions. These positions are mostly used for error and warning messages.

The third building block of a compiler, the context dependent analyzer or CDA for short, adds some more details to the nodes in the parser tree. In general these details can be found somewhere else in the parser tree, i.e. in the context, hence the name 'context dependent'. An example of such type of information is the data type of an identifier. This kind of information can be found by locating the declaration of the identifier, which in normal procedural languages has to precede the use of the identifier in a statement.

After the first three blocks of the compiler the parser tree has been built and it has been decorated with attributes. For all this the context free syntax of the source language is sufficient. Only the last part of the compiler, the code generator, requires the knowledge of the semantics of the source language. In other words, only the code generator has the capability of interpreting the meaning of the source text. For mapping the source code to the target code, the code generator also needs to know the semantics of the target language and of course also its syntax.

2.3.2 Building the TL to VHDL compiler using the GMD-toolbox

For building the TL to VHDL compiler the GMD-toolbox (GMD stands for Gesellschaft für Mathematik und Datenverarbeitung) from the university of Karlsruhe is used. This toolbox greatly simplifies the tremendous job of building a compiler. Using this toolbox, the tedious work of building the first two blocks of a compiler is almost completely done automatically. All the toolbox needs for this task is a specification of the concrete syntax of the source language, the structure of the parser tree (also called the abstract syntax), and the mapping of the concrete to the abstract syntax.

After the parser has finished its job, the context dependent analysis is started. This is done by walking through the parser tree, gathering the desired information along the way. Here the toolbox assists again (with the program 'Puma') by recognizing parts of the parser tree, sometimes covering several nodes. The specification for the CDA thus consists of a rule, which specifies the tree structure to be recognized, and the actions to be taken when a match is found. The actions are written in the language C.

One of the main things computed by the CDA is an environment tree. This environment tree is used to register where all identifiers are declared in the parser tree. The environment tree is not absolutely necessary, because the same results can be achieved by searching the parser tree for the declaration of the identifier. However the latter method is not by long as efficient as the environment tree, which uses an (unbalanced) ordered binary tree to store the identifiers. Chapter 4 will further elaborate on this.

Finally the code generator uses all the previously gathered information to generate the VHDL translation. The GMD-toolbox also offers programs helping in this, but these are directed at generating assembler code and they hardly support the task to be done by the TL to VHDL compiler. Instead the program Puma is used here again. Note that the part
of the TL specification that is also known by VHDL, can be copied unchanged to the VHDL specification.

The tactic used by the code generator is to open the source file again and then starting to walk the parser tree again, following the order of the source file. At some point during the traversal of the parser tree a node will be found representing something that has to be translated. The code generator copies all the text in the source file until the position of the TL specific code is reached. Then the code generator outputs two dashes to start the comment, copies the TL specific code from the source file and then starts generating the VHDL code on a new line. The new line is needed to finish the previously started comment.

Figure 2.6 gives an overview of the files used to generate the TL to VHDL compiler and the tools from the toolbox that do the job. The leftmost files contain the input specifications of the compiler and the rightmost files are the C source files generated by the tools. One of the major attractions of the toolbox is the tool 'Cg'. This tool extracts from the concrete syntax specification a large part of the scanner specification and it uses the same specification together with the specification of the concrete on abstract syntax mapping to extract the specifications for the parser. The tool 'Rpp' joins the two partial scanner specifications into one file and 'Rex' generates from that file the C source files for the scanner. 'Lair' uses the parser specification to generate the C files for the parser and a file with routines that implement a simple error recovery scheme for the parser.

From the abstract tree specification the tool 'Ast' generates an extensive set of routines to support building, traversing and transforming the specified tree. This tool is used to specify the parser tree as well as the environment tree. The routines generated for the parser tree are used by the parser and the CDA, whereas the routines for the environment tree are used exclusively by the routines that give access to the that tree.

The specifications of the CDA, the code generator and the routines giving access to the environment tree are translate by the tool Puma to C files. To recognize the specifications Puma needs to know the structure (attributes and children) of the nodes of the corresponding tree. This information gets Puma from the '.TS' file generate by Ast.

The file 'output' contains some routines to assist the code generator (e.g. routines to copy a part of the source file to the output). The 'main' file contains the user interface (only command line options) of the compiler and it calls the successive parts of the compiler, which are: the parser, the CDA and the code generator. Note that the scanner and error recovery routines are called by the parser and the environment access routines are called by the CDA. All the C source files are compiled and linked by the C compiler, which results in an executable TL to VHDL compiler.
Fig. 2.6: GMD-Toolbox overview.
In the next chapters only the actions of the CDA are concisely discussed. The specification of the scanner and the parser, are part of the front end of the compiler and thus not discussed in this report. See the report of O. Pigmans [Pig] for a full explanation on this part of the compiler. The implementation of the code generator is relatively straightforward since then all the required information has been made available at the right place by the CDA, therefore the code generator is not considered worth mentioning in the sequel.
3 VHDL IMPLEMENTATION OF TL CONSTRUCTS

The first section of this chapter states the main guidelines for the TL to VHDL translation. The other three sections of this chapter describe the syntax, semantics and VHDL implementation of the three TL constructs that currently can be compiled. These three constructs are events, mutual exclusion and communication.

3.1 General concepts for TL to VHDL translation

The requirements for the VHDL implementation of the TL design are that it has to be correct (obviously), comprehensible and fast. Consequently it is tried to replace the TL statements by subprogram calls and components and the TL objects by as little as possible signals. The advantage of using subprograms is that the VHDL code remains small and comprehensible and the compiler has little work to do. Moreover when the implementation has to be adjusted then it might be sufficient to change the VHDL code of the subprogram, so that there is no need to adapt the compiler.

The semantics of a TL statement prescribes the functionality of the VHDL code that implements it. Therefore only the design of the data construct of the object on which this statement operates can influence the clarity and efficiency of the VHDL code. One way to do this is keeping the number of signals that model a TL object low.

Implementing the TL objects in VHDL requires some type definitions (e.g. for a signal modelling an event object) and subprograms. The package defining these types and subprograms, called 'TL_TYPES', is explicitly made visible to every entity in the VHDL implementation file. The package 'TL_TYPES' also defines a constant 'delay', this constant is used in a wait-statement following the implementation of each TL specific statement. This so-called unit delay is the time it takes to execute a TL statement. The total time needed to finish a TL statement is that unit delay plus the time the statement is held up by one or more other processes.
3.2 Events

3.2.1 Syntax and semantics of TL events

An *event object* and the related statements provide a mechanism to synchronize two concurrent processes. The syntax of the declarations and statements related to this TL object is as follows:

- **event_dec** ::= EVENT event_list;
- **sig_stat** ::= SIGNAL_EVENT event_list;
- **wait_stat** ::= WAIT_EVENT event_expr;

(event_expr is a boolean expression with event objects)

The event declaration (event_dec) can appear in an interface declaration of an entity or a procedure, or in the declaration part of the architecture body. Both the statements can be used in the statement part of a leaf-task, a process or a procedure. Text 3.1 shows a small example using events.

```
ARCHITECTURE event_arch OF event_ent IS
  ...
  EVENT ev1, ev2, ev3;
BEGIN
  proc1: PROCESS
  BEGIN
    SIGNAL_EVENT ev1, ev2;
    ...
  END PROCESS;
  ...
  proc2: PROCESS
  BEGIN
    WAIT_EVENT ev1 AND ev3;
    ...
  END PROCESS;
...
proc3: PROCESS
BEGIN
  ...
  task3: TASK
  BEGIN
    WAIT_EVENT ev2;
  END TASK;
  ...
  task4: TASK
  BEGIN
    SIGNAL_EVENT ev3;
  END TASK;
  ...
END;
```

Text 3.1: Example demonstrating the usage of events.

---

1 For the syntax definitions the following notation is used: `::=` means is defined as, `{A}` means A repeated zero or more times, `[A]` means A is optional and A | B means A or B. Words written in capitals are keywords from the TL language. Words in lower case are so-called none-terminals.
The event synchronization should be modelled using the split binary semaphore technique [Dij, Ray]. Normally the signal_event-statement changes the event object (or objects) to 'true' and then continues with the next statement. However when a signal_event-statement is reached again (thus not the first time) then the signalled process may not have processed the previous signal. In such a case the signalling process has to wait until the previous signal has been processed by the signalled process. When the signal-statement with more than one event is executed not for the first time and one of the events has not been processed by the signalled process, then this may not prevent the other events in the statement from being signalled.

The wait_event-statement normally waits till the event expression becomes true, unless the expression was already true to start with. Then the wait_event-statement changes the value of the involved event objects to 'false' before it continues with the next statement. The involved event objects may be reset (made 'false') no sooner than that the event expression is true. Note that for each event object named in the boolean event expression of the wait_event-statement there has to be a signal_event-statement signalling that event object and vice versa.

### 3.2.2 Implementation of events

#### 3.2.2.1 Event object implementation

The event object is implemented using one signal, representing its boolean value, with a resolution function. A resolution function is compulsory because both the signal_event-statement and the wait_event-statement assign a value to this signal so the signal has two drivers. The signalling process always assigns the value true to this signal (only when the event is false) while the waiting process always assigns the value false (only when the event is true). This means that the driver of the signalling process always emits true while the driver of the waiting process always emits false. These two values are presented to the resolution function that has to determine what value the signal should get.

What the resolution function needs to know is which driver made the most recent assignment. This is achieved by making the signal a record of two boolean fields. One field, called 'value', always reflects the status of the event and the other field, called 'private', is used by the signalling and the waiting process to let the resolution function know who made the last assignment to the event. Table 3.1 shows the values received and returned by the resolution function. The first row displays the initial situation in which neither the wait_event-statement nor the signal_event-statement has assigned a value to the signal. In this case both the records are equal so returning either of them is good.

The first statement that can be executed is always the signal_event-statement because the event is initially false. Since the signal_event-statement always assigns true to the value field of the record the resolution function can now always see which record
Table 3.1: Functional description of the resolution function for event signals.

<table>
<thead>
<tr>
<th>signal statement</th>
<th>wait statement</th>
<th>result</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>private value</td>
<td>private value</td>
<td>private value</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>initial</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>signal statement executed</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>wait statement executed</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>signal statement executed</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>wait statement executed</td>
</tr>
</tbody>
</table>

comes from the signalling process. The private field of the signalling process has to indicate that the signalling process did the last assignment, therefore it has to change value. The same tactic is used by the waiting process to signify that it made the most recent assignment. Eventually both the statements keep toggling the value of their private field each time the statement is executed. The resolution function uses both the private fields to calculate the new status of the event (using an exor) and thus to conclude which driver made the most recent assignment.

The resolution function has to consider one other possibility. This is the case when there is only one driver. This case occurs in a hierarchical system where the signalling and waiting process are located in a different entity. In this case the resolution function has to return the only record it gets from the simulator. Text 3.2 exhibits the parts of the 'TL_TYPES' package declaration and package body declaring the resolution function and other definitions used to implement event objects.

Note that VHDL has a means (using attributes) of detecting the time elapsed since the last assignment was done to a signal. Unfortunately the array offered to the resolution function by the simulator is a variable, not a signal and for variables there is no way to detect which driver made the most recent assignment. So this VHDL feature cannot be used to simplify the event object implementation.

A major advantage of the implementation of the event object with one signal (as opposed to the implementation with two signals [Boel]) is that there no translation needed for the interface associations of procedure calls and component instantiations. Next to that an implementation with one signal also keeps the expressions (used by the event statements) to test the status of the event simple.
PACKAGE tl_types IS
  ... **** EVENTS ****
  TYPE event_record IS RECORD
    private: BOOLEAN;
    value: BOOLEAN;
  END RECORD;
  TYPE event_record_array IS ARRAY(NATURAL RANGE 0) OF event_record;
  FUNCTION event_record_func (drivers: IN event_record_array) RETURN event_record;
  SUBTYPE event_type IS event_record_func event_record;
END tl_types;
PACKAGE BODY tl_types IS
  ... **** EVENTS ****
  FUNCTION event_record_func (drivers: IN event_record_array) RETURN event_record IS
    VARIABLE sig: NATURAL;
    BEGIN
      IF drivers'LENGTH = 2 THEN
        -- identify driver
        IF drivers(0).value
          THEN sig := 0;
          ELSE sig := 1;
        END IF;
      END IF;
      -- determine event value
      IF drivers(0).private XOR drivers(1).private
        THEN RETURN drivers(sig); -- event is just made true, return signal statement value
        ELSE RETURN drivers(1-sig); -- event is just made false, return wait statement value
      END IF;
      END IF;
      RETURN drivers(drivers'LEFT); -- one driver, pass it on
    END event_record_func;
  END tl_types;

Text 3.2: The part of the "TL_TYPES" package used for events.

3.2.2.2 Event statements implementation

According to its semantics the signal_event-statement has to check first whether or not a previous signal has been processed (i.e. the event is false). If this is not the case then the signalling process has to wait until the corresponding waiting process executed the wait_event-statement. Notice however that a signal_event accepts a list of events to be signalled. When in such a case the same signal_event-statement was executed before, then semantics demands that the events that meanwhile have been processed by their corresponding waiting process are immediately signalled again. Events that are not yet processed have to be signalled as soon as the previous signal of the event is processed.

This all results in a loop-statement that contains for each event to be signalled statements to signal the event when it is false. Because the loop can be executed several times care has to be taken that an event can only be signalled once. This is achieved with a boolean variable that records whether or not the event has been signalled. The loop is ended via an exit-statement when the variables indicate that all the events to be
signalled have been signalled. The loop-statement also contains a wait-statement that suspends the signalling process when it has to wait for one or more previously signalled events to be processed. This wait-statement is required else the simulator would never stop executing the loop-statement, and thus the simulation would never continue.

Regrettably the above described implementations cannot be replaced by calling a general procedure. This is caused by the fact that such a procedure has to be able to handle an arbitrary number of events to be signalled. This can only be done by using an unconstrained array in the interface declaration of the subprogram. To use such a subprogram, the unconstrained formal parameter has to be associated to a constrained actual parameter (as opposed to associating each element of the array separately). Therefore all event declarations should be gathered in the arrays according to the event lists of the signal_event-statements. However this is not possible since some events have to be declared in the interface of the entity while others have to be declared in the declaration part of the architecture body. The only alternative is that the statements are inserted in-line, i.e. replace the signal_event-statements.

The implementation of the wait_event-statement first has to wait until the event expression becomes true. This is done with a wait-statement in an if-statement. The condition of this if-statement is the reverse of the event expression so that the wait-statement is only executed when the expression turns out to be false when it is first checked. It is necessary to use the if-statement because the VHDL wait-statement always start with waiting for an action (assignment by another process) to occur on any of the signals involved in the expression. Only after an action has occurred, it is checked if the expression evolves to true so that the next statement can be executed. So when the expression is initially true then the wait-statement still suspends the process waiting for an assignment to any of the signals.

When the event expression proves to be true, all the involved events that are true are made false again. Note that an OR in the event expression allows that not all events in the expression are true. Also note that the wait_event-statement makes events false no sooner then that the event expression has evolved to true. This in contrast to the signal_event-statement where an event is signalled independent of other events to be signalled by the same signal_event-statement (if any). This is a consequence of the semantics of the wait_event-statement.

Due to the event expression the wait_event cannot be translated by a subprogram call. The event expression has to be passed to the subprogram unevaluated and that feature is not available in VHDL (or any other procedural language). Text 3.3 lists the result of translating text 3.1 using the above explained translations. Note that in the translation the keywords and identifiers for the leaf-task-statement are simply put in comment and don't need to be replaced by VHDL code. This is because the leaf-task-statement merely serves as a way to group statements, which is desired for automatically analyzing the design.
ARCHITECTURE event_arch OF event_ent IS

-- EVENT ev1, ev2, ev3;
SIGNAL ev1, ev2, ev3: event_type;
BEGIN
proc1: PROCESS
  VARIABLE ev1_var: BOOLEAN;
  VARIABLE ev2_var: BOOLEAN;
  BEGIN
    SIGNAL_EVENT ev1, ev2;
    ev1_var := FALSE;
    ev2_var := FALSE;
    LOOP
      IF NOT ev1_var AND NOT ev1.value THEN
        ev1 <= (NOT ev1.private, TRUE);
        ev1_var := TRUE;
      END IF;
      IF NOT ev2_var AND NOT ev2.value THEN
        ev2 <= (NOT ev2.private, TRUE);
        ev2_var := TRUE;
      END IF;
      EXIT WHEN ev1_var AND ev2_var;
      WAIT ON ev1, ev2;
    END LOOP;
    WAIT FOR delay;
  END PROCESS;
proc2: PROCESS
  BEGIN
    WAIT_EVENT ev1 AND ev3;
    IF NOT (ev1.value AND ev3.value) THEN
      WAIT UNTIL ev1.value AND ev3.value;
    END IF;
    IF ev1.value THEN
      ev1 <= (ev1.private, FALSE);
    END IF;
    IF ev3.value THEN
      ev3 <= (ev3.private, FALSE);
    END IF;
  END PROCESS;
proc3: PROCESS
  VARIABLE ev3_var: BOOLEAN;
  BEGIN
    SIGNAL_EVENT ev3;
    ev3_var := FALSE;
    LOOP
      IF NOT ev3_var AND NOT ev3.value THEN
        ev3 <= (NOT ev3.private, TRUE);
        ev3_var := TRUE;
      END IF;
      EXIT WHEN ev3_var;
      WAIT ON ev3;
    END LOOP;
    WAIT FOR delay;
  END PROCESS;
END;

Text 3.3: VHDL translation of text 3.1.
3.3 Mutual exclusion

3.3.1 Syntax and semantics of TL mutual exclusion

In order to model the behaviour of an embedded system, it might be necessary to make some data structures accessible to more than one process. When those processes simultaneously access the data structure this can result in corrupt data being read or written. The TL language offers a mutual exclusion protection mechanism to prevent these errors. The mutual exclusion mechanism ensures that the data structure can never be accessed by more than one process at once. Note that the syntax definition of TL does not allow data to be accessed by several processes without using the mutual exclusion mechanism. The syntax of the declarations and statements related to this TL object is as follows:

```
me_data_dec ::= ME_DATA me_data_name_list: subtype_indication
               [:= initialize_expression];
               (subtype_indication defines the type of the data structure)
               (the optional initialize_expression defines the initial value)

me_data_name_list ::= me_data_name {, me_data_name}

wait_exclusion_stat ::= WAIT_EXCLUSION me_data_name;

release_exclusion_stat ::= RELEASE me_data_name;
```

The mutual exclusion declaration (me_data_dec) can appear in an interface declaration of an entity or procedure and in the declaration part of the architecture body. Both the statements can be used in the statement part of a leaf-task, a process or a procedure. Text 3.4 shows a part of a TL design using a mutual exclusion object.

The mutual exclusion object (the data structure) itself should be modelled with a signal. The mutual exclusion protection algorithm has to prevent deadlock and starvation. A deadlock can occur when several processes are simultaneously requesting access to the same object. Deadlock means that in such a situation every process infinitely waits, e.g. until the other requesting processes are finished with the object. Preventing starvation implies that a process that requests access to the object is sure to get that access at some time in the future, as opposed to waiting indefinitely. Preventing starvation is also known as ensuring fairness.
3.3.2 Implementation of mutual exclusion

The mutual exclusion object consists of two parts, namely the data structure and the mutual exclusion algorithm protecting that data structure. These parts are implemented independent from each other and thus are also discussed separately in the next two sections.

3.3.2.1 Mutual exclusion data structure implementation

The implementation of the mutual exclusion data structure has to permit that the object is read and written by several processes. This suggests that those processes are interconnected by means of a data bus. The mutual exclusion protection of the data structure guarantees that there will never be more than one process accessing the object at any time. All processes not allowed to access the mutual exclusion object have to be disconnected from the bus. However in section 2.1.5 it was theorised that the VHDL model for such a bus cannot be used.

The problem only occurs when there is an entity connected to the data bus in which at some time there is no driver for the bus. From this can be derived that a way to avoid the problem would be to make sure that the drivers never disconnect from the bus. Instead the drivers that should be disconnected have to drive some special value that indicates to the resolution function that the driver should be ignored. This special value has to applicable to any arbitrary type, even new types defined by the designer.

The implemented solution combines the original data type of the mutual exclusion object with a new special type in a record. The special type merely flags if a driver is connected, in which case the other field contains the value of the object. Or it flags that the driver is disconnected, in which case the other field should be ignored. If the flags of all records received by a resolution function indicate that the corresponding driver is
'disconnected' then resolution function has to return a record that indicates that the entity as a whole is disconnected. In the sequel this databus model is referred to as the 'bus emulation'.

The declarations for this bus emulation, i.e. a record type definition and a resolution function for each type used for mutual exclusion (see text 3.5), are inserted in a special packages, named 'special_types', which is added at the top of the design file. The contents of the 'special_types' package is made visible to all the design units. In section 3.4.3 this package, which contains many more declarations, is discussed in full.

```
-- the 'id_type' type definition is inserted in the 'TL_TYPES' package
TYPE id_type IS (disconnected, connected, store);

-- the next declarations are inserted in the 'special_types' package
TYPE <TypeName>_record IS RECORD
  data: <TypeName>;
  id: id_type;
END RECORD;

TYPE <TypeName>_record_array IS ARRAY(NATURAL RANGE 0) OF <TypeName>_record;

FUNCTION <TypeName>_resolve (driver: IN <TypeName>_record_array) RETURN <TypeName>_record IS
BEGIN
  temp:=driver(driver'LEFT);
  FOR i IN driver'RANGE LOOP
    IF driver(i).id = connected THEN RETURN driver(i); END IF;
    IF driver(i).id = store THEN temp:=driver(i); END IF;
  END LOOP;
  RETURN temp;
END;

SUBTYPE <TypeName>_bus_type IS <TypeName>_resolve <TypeName>_record;
```

Text 3.5: Declarations for bus emulation.

Next to the data bus, the implementation for the mutual exclusion object also needs a register to store the value of the object. The value in this register is only significant when all drivers in the design are disconnected from the data bus. This register behaviour is modelled by an extra process. This extra process, called the store, which is added at the highest hierarchical level, always copies the old value of the signal back to the signal. This creates a driver for the resolution function on the highest hierarchical level that always drives the previous value. When all other drivers are 'disconnected' then the resolution function returns this old value, thus modelling the memory function of the object. To do this properly the resolution function needs to be able to recognize the special driver, because the old value has to be ignored when there is another driver connected. Therefore the store changes the flag to the value 'store' to indicate that it drives the old value.

The declaration of the mutual exclusion object can specify an initial value. In the translation the whole expression that specifies the value is incorporated in an initialization of the record, thus including the flag. The value of the flag used for this
purpose has to be 'disconnected' because it is the initial value for all drivers in the system.

When the above described implementation of the mutual exclusion object is compared with the previous implementation [Boel] (the one for hierarchical systems), this new implementation is notably more elegant. This is due to the fact that in this first attempt the bus emulation was not recognized. To get around the VHDL bus problem this first implementation copies the value of the mutual exclusion store to a register which is private to the process that is granted access to the object. Using this implementation would allocate a great deal of the resources for a number of registers which are hardly ever used, which of course is not desirable.

3.3.2.2 Mutual exclusion algorithm implementation

The mutual exclusion algorithm is the algorithm that determines which of the processes claiming access to the mutual exclusion object will actually get that access. For software applications several commonly known mutual exclusion algorithms are fully researched [Ray]. All of them of course prevent deadlock and guarantee fairness. One of those algorithms is chosen to implement mutual exclusion for the TI mutual exclusion object. Another algorithm is also a potential candidate but it is not implemented yet, this algorithm is explained in section 5.2.

The implemented algorithm uses an extra concurrent process that supervises the access to the mutual exclusion object. This process, the arbiter, receives all requests to access the object from all the processes using the object. For this purpose each process reports its status to the arbiter via a private status-signal. The arbiter is located at the highest hierarchical level in which the mutual exclusion object appears, which is next to the store process described in the previous section.

The arbiter receives the status of all processes in an array of status-signals. The arbiter scans this array for a process requesting access, the first process found is granted permission to access the object. When this process releases the object the arbiter withdraws its grant and starts again to scan the whole array to see if any process wants to use the object. To ensure fairness the scan has to start from the successor of the process that just accessed the object (the successor of the last signal in the array is the first signal in the array). When it turns out that none of the processes is requesting access, the arbiter waits (suspends) until a process makes a request. This wait is obligatory in order to avoid the arbiter procedure to be executed indefinitely.

The arbiter is independent of the type of the mutual exclusion object it protects, this together with the fact that VHDL knows unconstrained arrays allows that the function of the arbiter can be implemented using a procedure. Text 3.6 shows the VHDL procedure implementing the arbiter.

The status-signals of the processes are used by the process to ask the arbiter for access to the object. The arbiter uses the same signal to grant that access. As soon as the process has finished, it uses the signal to report that the object is free again. The type of
PROCEDURE arbiter (SIGNAL channels: INOUT lTl8_status_type_arr) IS
  VARIABLE previous: NATURAL; -- number of last active process
BEGIN
  previous:=channels'HIGH; -- initially test all processes
  LOOP
    FOR i IN channels'RANGE LOOP
      IF channels(i)=request THEN -- request found
        channels(i)<=grant;
        WAIT UNTIL channels(i) = free;
        previous:=i; -- make a full round
        NEXT;
        IF i-previous THEN -- made a full round, no request found
          WAIT ON channels;
        END IF;
      END IF;
    END LOOP;
  END LOOP;
END;

Text 3.6: The VHDL implementation of the arbiter.

The status-signal is a special type that only knows the three above named values, i.e. 'request', 'grant' and 'free'. Since both the arbiter and the process are driving the status-signal there is a resolution function required. The resolution function either gets one or two drivers. As for the resolution function for events the one driver situation can occur only when the two drivers are located in two different entities. This resolution function also simply has to return the only value available in that situation. In the case of two drivers the resolution function has to react as is listed in table 3.2. Text 3.7 shows the definitions used for the mutual exclusion status type. These definitions together with the definition of the arbiter procedure are included in the 'TI.._TYPES' package.

<table>
<thead>
<tr>
<th>process</th>
<th>arbiter</th>
<th>channel</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>free</td>
<td>request</td>
<td>process request access</td>
</tr>
<tr>
<td>request</td>
<td>grant</td>
<td>grant</td>
<td>arbiter grants access</td>
</tr>
<tr>
<td>free</td>
<td>grant</td>
<td>free</td>
<td>process frees object</td>
</tr>
<tr>
<td>free</td>
<td>free</td>
<td>free</td>
<td>arbiter withdraws grant</td>
</tr>
</tbody>
</table>

Table 3.2: Specification of the resolution function for mutual exclusion status-signals.

The current mutual exclusion algorithm is equal to one of the two algorithms proposed by M. Boersma [Boel]. However by using a concurrent procedure call instead of in-line code the current implementation reduces the generated VHDL code considerably. The other algorithm proposed by M. Boersma proves to be less suitable for hierarchical designs. This mutual exclusion algorithm lets the processes requesting access determine...
Text 3.7: Definitions for the mutual exclusion status-type.

them self whether it is their turn to get access. To do this, every process using a mutual exclusion object has to be able to see the status of all other processes using that object. The problem is that the signals carrying the status of the processes not only run upwards in the hierarchy (as is the case for the current implementation), but also downwards (see fig. 3.1). This implies that a small entity, containing one process using the object, needs in its interface an array of N signals to attach the status-signals to, where N can be arbitrarily large. Also this entity can be instantiated more than once and thus be joined together with more than one object. This means that N cannot be fixed. Although VHDL allows this (GENERIc), it does result in a rather complex design.

![Fig. 3.1: Distributed mutual exclusion algorithm.](image_url)

Note however, that this implementation is suitable for a software implementation, where the array of signals can be declared globally. In a software implementation the algorithm has the pleasant property that there is no task switching required when a process asks for permission to access the mutual exclusion object.
3.3.2.3 Mutual exclusion statements implementation

The mutual exclusion statements are relatively simple because they only allow one mutual exclusion object as an operand. Due to their simplicity both the statements could be implemented in a procedure. Alas the statements need to handle the object itself so that the procedures have to use the data type of the object. That is also the reason why the declarations of the procedures are added to the 'special_types' package that also declares bus emulation declaration for that type.

The wait_exclusion-statement first indicates to the arbiter that it wants access to the object and then it waits until the arbiter grants permission to the object. Subsequently the driver for the mutual exclusion object is connected to the data bus so that the resolution function knows where the only relevant value is coming from. Note that next to assigning the right value to the flag, it is also important to copy the value of the object. If this is not done then the driver will keep driving the same value as before, which is probably totally different from the current value of the object.

The release-statement is even simpler than the wait_exclusion-statement. First a wait-statement is needed to make sure that the last value assigned to the mutual exclusion object has reached the store. Leaving out this wait-statement would combine this last value with a 'disconnected' flag and the value would be ignored. Next the driver is disconnected from the bus implementing the mutual exclusion object and then it is indicated to the arbiter that the object is no longer needed.

Note that the unit delay (see section 3.1) here is needed for correctness. This is because a wait_exclusion-statement for the same object could be immediately following the release-statement. Without the wait-statement the status-signal of the process would be changed at once to 'request' while the arbiter would not have had a chance to withdraw its grant. From table 3.2 it can be seen that the resolution function will return a grant. So the wait_exclusion-statement would mistakenly 'think' it was granted access even before the new request reached the arbiter.

Text 3.8 contains the bodies of the procedures implementing the mutual exclusion statements. Note that the release-statement actually does not need to know the type of the object as it only handles the flag. Text 3.9 lists the translation of the example from text 3.4.
PROCEDURE <TypeName>_wait_excl (SIGNAL req-chan: INOUT me_status_type;
    SIGNAL data_bus: INOUT <TypeName>_bus_type) IS
BEGIN
    req-chan <= request;
    WAIT UNTIL req-chan = grant;
    data_bus <= (data_bus.data, connected);
    WAIT FOR delay;
END;

PROCEDURE <TypeName>_release (SIGNAL req-chan: INOUT me_status_type;
    SIGNAL data_bus: INOUT <TypeName>_bus_type) IS
BEGIN
    WAIT FOR 0 NS;
    data_bus.id <= disconnected;
    req-chan <= free;
    WAIT FOR delay;
END;

Text 3.8: The procedures implementing the mutual exclusion statements.

ARCHITECTURE mutex_arch OF mutex_ent IS
    COMPONENT comp1
        PORT (-- ME DATA mem: memory;
            mem_me_ch: INOUT me_status_type_arr (0 TO 0);
            mem_bus: INOUT memory_bus_type);
        END COMPONENT;
    -- ME_DATA ram: memory := (0, 1, 2);
    SIGNAL ram_me_ch: me_status_type_arr (0 TO 1);
    SIGNAL ram_bus: memory_bus_type := (0, 1, 2), disconnected;
    ALIAS ram: memory IS ram_bus.data;
BEGIN
    inst1: comp1 PORT MAP (-- mem => ram
        mem_me_ch => ram_me_ch(0 TO 0),
        mem_bus => ram_bus);
    proc1: PROCESS
        BEGIN
            -- task1: TASK
            -- BEGIN
            -- WAIT EXCLUSION ram;
            memory_wait_excl (ram_me_ch(1), ram_bus);
            -- exclusive access to ram
            ...
            -- END TASK;
        -- task2: TASK
        -- BEGIN
        -- RELEASE ram;
        memory_release (ram_me_ch(1), ram_bus);
        ...
            -- END TASK;
        END PROCESS;
    ram_arbitr: arbitrator(ram_me_ch);
    ram_store: ram_bus <= (ram, store);
END;

Text 3.9: Translation of the example in text 3.4.
3.4 Communication

3.4.1 Syntax and semantics of TL communication

Next to synchronization and the mutual exclusion, TL offers one other important mechanism through which tasks can collaborate to complete a computation. This mechanism, called communication, enables tasks to exchange messages. TL does not allow broadcasting, it only allows point to point communication. To enforce this rule TL demands that the communication partners are explicitly identified in a communication action. In return TL guarantees that the communicated data cannot get lost, not even when one of the communicating partners gets temporarily suspended. To transport the communicated data from one task to another, TL uses channels. A channel may be utilized as a transport medium for several communication actions. The syntax of the declarations and statements related to this TL object is as follows:

\[
\text{protocol\_dec} ::= \text{PROTOCOL protocol\_name\_list: protocol\_type\_name};
\]
\[
\text{protocol\_name\_list} ::= \text{protocol\_name}\{, \text{protocol\_name}\}
\]
\[
\text{protocol\_type\_name} ::= \text{HANDSHAKE I HANDSHAKE\_MERGED I}
\text{RENDZVOUS I RENDZVOUS\_MERGED (the currently available standard protocols)}
\]
\[
\text{interface\_channel\_dec} ::= \text{CHANNEL interface\_channel\_list: protocol\_name}
\]
\[
\text{interface\_channel\_list} ::= \text{interface\_channel}\{, \text{interface\_channel}\}
\]
\[
\text{interface\_channel} ::= \text{channel\_name ( subchannel\_list )}
\]
\[
\text{subchannel\_list} ::= \text{subchannel}\{, \text{subchannel}\}
\]
\[
\text{subchannel} ::= \text{source\_name -> dest\_name}
\]
\[
\text{channel\_dec} ::= \text{CHANNEL channel\_list: protocol\_name};
\]
\[
\text{channel\_list} ::= \text{channel\_name}\{, \text{channel\_name}\}
\]
\[
\text{com\_data\_dec} ::= \text{COM\_DATA message\_list: subtype\_indication}
\text{[:= initialize\_expression];}
\text{(subtype\_indication defines the type of the messages)}
\text{(the optional initialize\_expression defines the initial value)}
\]
\[
\text{message\_list} ::= \text{message\_name}\{, \text{message\_name}\}
\]
send_stat ::= SEND message_name TO task_list [ON channel_name];

task_list ::= task_name {, task_name}

receive_stat ::= RECEIVE message_name FROM task_name [ON channel_name];

The protocol declaration (protocol_dec) may be included in a package or in the declaration part of an architecture body. The interface channel declaration (interface_channel_dec) can appear in an interface declaration of an entity or procedure. The channel declaration (channel_dec) can only appear in the declaration part of the architecture body. The communication data declaration, the message object, can be found in the declaration part of a process-statement or a procedure. Communication statements are identified by the label of the leaf-task in which they occur, therefore these statements can only be used in the statement part of leaf-tasks. Text 3.10 shows a part of a TL design using the communication constructs.

```
ARCHITECTURE comm_arch OF comm_ent IS

COMPONENT comp1
  PORT (CHANNEL ch1 (tskA -> tskB));
END COMPONENT;

PROTOCOL prot1: HANDSHAKE_MERGED;
CHANNEL c:h2;
CHANNEL c:h3: prot1;
BEGIN
  inst1: comp1 PORT MAP (ch2 (tsk3->tsk6));
  proc1: PROCESS
    COM_DATA msg1: msg_type;
    BEGIN
      tsk1: TASK
        BEGIN
          RECEIVE msg1 FROM tsk4 ON ch3;
        END TASK;
      ... 
      tsk2: TASK
        BEGIN
          SEND msg1 TO tsk5 ON ch3;
        END TASK;
    END PROCESS;

  proc2: PROCESS
    COM_DATA msg2: msg_type:= expr1;
    COM_DATA msg3: msg_type;
    BEGIN
      tsk3: TASK
        SEND msg2 TO tsk6 ON ch2;
        msg2 <= expr2;
        END TASK;
      ... 
      tsk4: TASK
        SEND msg2 TO tsk1 ON ch3;
        ... 
      tsk5: TASK
        RECEIVE msg3 FROM tsk2 ON ch3;
        ... 
    END PROCESS;
END;
```

Text 3.10: Example demonstrating the usage of communication.

The message objects should be modelled with variables. Operations on this object do not affect the data previously sent. The subtype of the message used in a send-statement must be equal to the subtype of the message used in corresponding receive-statement. Note that TL only allows message objects to specify the data to be transferred.
Between each pair of tasks communicating with each other, i.e. each point to point communication, a subchannel and a channel are allocated (see fig. 3.2). The subchannel will be used to synchronize the communication partners and the channel carries the message. All the channels used by one send-statement are mapped onto one multidrop channel (a channel with possibly more than one reader). The subchannels to be used in local communication action cannot be declared in TL, hence they are always implicit. This in contrast to the subchannels used in a none local communication action, for which the subchannel has to be declared explicitly in the interface channel declaration of the entity or procedure. Note that TL allows none local channels to be used for local communication actions.

3.2: Point to point communication with a channel and subchannels.

For channels declared without a protocol specification, a default protocol will be used. If a channel is used by more than one send-statement then that channel is said to be a merged channel. For a merged channel the declaration has to explicitly specify a merged protocol. A merged protocol is a protocol that incorporates a mutual exclusion protection algorithm to protect the channel. This assures that the channel can be used by several senders, while still ensuring the integrity of the data sent over the channel.

The protocol for a channel is implemented in the communication statements using that channel and sometimes by one or more dedicated processes for that channel (e.g. an arbiter). Naturally the implementation of a TL communication statement is determined by the protocol of the used channel. On the other hand the type of the channel is determined by the type of the messages being sent over the channel. This implies that all messages sent over one channel must have a similar type. More precisely, the types of those messages must all be a subtype of the same type.

For a communication action for which no channel is specified, an implicit channel with a default protocol will be used. For such a case the communication action has to be local (i.e. communication between processes in the same entity). Note that an implicit channel can never be a merged channel.

When a send-statement is executed not for the first time, then this statement can only proceed with sending the new message if all the receivers have accepted the previous message. A send-statement implementing a merged protocol naturally has to wait when another communication action is taking place on the channel. If one receiver of a set of receivers corresponding to one send-statement does not accept the offered data, then this
may not block the other receivers that already have accepted that data. Unless of course these receivers want to accept the next message.

The TL language allows the protocol definition to be incorporated into the specification, which enables the designer to specify own protocols. However, this feature is not yet supported by the compiler. For now the compiler only supports the four standard protocols named in the syntax definition, which currently are only specified outside TL. For the handshake protocol the channel behaves like a register. This means that when the channel is free the send-statement can put its data on the channel and subsequently the task can go on with the next statement without having to wait for the receiver to accept the data. The receiver for the handshake protocol waits until there is data on the channel, copies the data to the communication data variable and then acknowledges the sender that the data has been received.

For channels that do not behave like a register, the rendezvous protocol should be used. The rendezvous protocol ensures that the tasks involved in the communication action _lockstep_, i.e. that they simultaneously perform the communication action. Therefore if one of the two communication partners arrives first it has to wait until the other has arrived too (see fig. 3.3). As soon as both the communication partners have arrived, the data is transferred. When the receiver has acknowledged that it has received the data then both tasks can proceed. If there are several receivers then the sender has to wait till the last receiver has finished. However this should not keep other receivers that have accepted the message from proceeding, unless they want to accept the next message.

![Diagram of the rendezvous protocol](image)

**Fig. 3.3:** The rendezvous protocol.

### 3.4.2 Implementation of communication

The previous TL to VHDL compiler [Boe1] only allowed the handshake protocol, therefore only the implementation for this protocol can be compared with this previous implementation. The first compiler allowed the communication to be buffered, this in
contrast to the current compiler, however some ideas for buffering are discussed in section 5.3.2.

The next sections describe the implementation of the communication constructs for each protocol type, because the implementation of the communication constructs is dominated by it. The only two things not influenced by the protocol are the implementation of the message object, which is simply implemented by a VHDL variable, and the protocol declaration, which is reduced to comment. The last section describes the solution of a problem for types used for the communication (this problem was not considered in the first compiler).

3.4.2.1 Handshake protocol implementation

The handshake protocol can be used when there is only one sender using the channel; therefore the channel can be modelled by a single signal. The type of the signal depends on the data type of the message sent over the channel. If the channel is declared in the entity interface (interface_channel_dec) then the direction of that signal has to be determined. For each subchannel associated with the channel, i.e. for each receiver, a signal is allocated. The name used for this signal is the concatenation of the name of the sender and the name of the receiver. The point to point signal indicates the status of the point to point communication, which is either 'data_sent' or 'data_received'. The value 'data_sent' is assigned to the signal by the sender and the value 'data_received' has to be assigned by the receiver. This means that the signal has two drivers and thus needs a resolution function. Since for events a similar signal and resolution function have already been designed, they are used here again. For this implementation the value 'true' models 'data_sent' and the value 'false' models 'data_received'.

The implementation of the send-statement first has to check if a previously sent message has been accepted by all of its receivers. If this is not so then the send-statement waits until the last receiver makes its subchannel false. At this point the channel is free and the sender copies the value from the variable modelling the message object to the signal that models the channel. Next all signals modelling a subchannel are set to indicate that new data is available and the sender-task can proceed executing the next statement.

The implementation of the receive-statement starts with waiting till its subchannel signifies that there is valid data on the channel. When the data is available on the channel, it is copied to the variable modelling the message object. Then this data reception is acknowledged to the sender by resetting the subchannel and the receiver-task can proceed.

The implementations for this protocol (see text 3.11) are almost equal to the implementation in the first TL to VHDL compiler. The only difference is that the implementation for the events has been optimized in the current compiler.
Text 3.11: Implementation communication statements for the handshake protocol.

3.4.2.2 Rendezvous protocol implementation

The declaration of the signal modelling the channel is exactly the same as the declaration of a channel with a handshake protocol. Again the subchannel is mapped onto one signal, only here a new type definition and resolution function is required. From figure 3.3, which shows the rendezvous protocol, it can be derived that the new type knows four values, namely 'data_present', 'acknowledge', 'no_data' and 'free'. The values from the two drivers (the sender and the receiver) have to be combined by the resolution function to produce one of the four values. This value has to reflect the value that was last assigned to the subchannel. Table 3.3 specifies the behaviour of the resolution function and text 3.12 list the declarations of the type and its resolution function. Note that in the initial situation both drivers drive the value 'free' (it is the first literal of the enumeration type definition), this is also the value that has to be returned by the resolution function.

<table>
<thead>
<tr>
<th>sender</th>
<th>receiver</th>
<th>subchannel</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_present</td>
<td>free</td>
<td>data_present</td>
<td>sender send data</td>
</tr>
<tr>
<td>data_present</td>
<td>acknowledge</td>
<td>acknowledge</td>
<td>receiver acknowledges</td>
</tr>
<tr>
<td>no_data</td>
<td>acknowledge</td>
<td>no_data</td>
<td>sender removes data</td>
</tr>
<tr>
<td>no_data</td>
<td>free</td>
<td>free</td>
<td>receiver removes acknowledge</td>
</tr>
</tbody>
</table>

Table 3.3: Specification of the resolution function for rendezvous subchannels.

For the rendezvous protocol the sender first has to check if the receivers have ended a possible previous communication action, i.e. it has to check if the subchannel has the values 'free' (see fig. 3.3). If the subchannels indicate 'free' then the send-statement can proceed by assigning the message data to the channel and assigning 'data_present' to all subchannels. Now the send-statement has to wait until all receivers have acknowledged that they have received the newly sent message, i.e. it has to wait until all the
Text 3.12: Definitions for the rendez-vous protocol subchannels.

subchannels are changed to 'acknowledge' by the receivers. Semantics demands that a
the sender assigns 'no_data' to the subchannel as soon as the corresponding receiver has
acknowledged the reception of the message. This enables the fast receivers to continue
while the sender may still be waiting for other (slow) receivers.

The implementation of the receiver waits first till the subchannel indicates
'data_present'. Then the receiver copies the message and reports that fact to the sender
by assigning 'acknowledge' to the subchannel. As soon as the sender has reacted on that
by returning 'no_data', the receiver can proceed by making the subchannel 'free' and
executing with the next statement (after a unit delay). Text 3.13 lists the implementa­
tions of the communication statements for this protocol.

3.4.2.3 Handshake-merged protocol implementation

The handshake-merged protocol is intended for channels that are used by several send­
statements to transfer their messages to maybe even more receivers. The protocol
incorporates a mutual exclusion algorithm to protect the channel. The mutual exclusion
algorithm is currently implemented by a same arbiter process as used to protect a
mutual exclusion object. This arbiter is also located at the highest hierarchical level
where the protected item, in this case thus the channel, is used. The access requests for
the channel will be done by the send-statement, therefore for each send-statement a
status-signal to the arbiter has to be reserved.
Text 3.13: Implementation communication statements for the rendezvous protocol.

For the implementation of the channel object the same considerations are in effect as for the implementation of the mutual exclusion object (see section 3.3.2.1). Namely several processes have to be permitted to read and write to it. A consequence is that the implementation of the channel object will use the same bus emulation scheme. Semantics demands that the channel with a handshake protocol behaves like a register, so even the store is required to implement the channel object.

The handshake merged protocol demands that a sender process can proceed with its task even if the receivers do not immediately accept the data. However, the merged channel which is used for this communication action is used by others too, so this channel has to be made available to these others as soon as it is no longer needed for this communication action. This leads to the conclusion that releasing the merged channel has to be done concurrently to the execution of the sender task. For this purpose an extra process is added for each send statement using this protocol, this process is called the 'Handshake Merged sender process' or 'HSM-sender process' for short.

The task of this HSM-sender process is to release the mutual exclusion access to the channel as soon as all receivers have accepted the data. In order to do this the HSM-sender process has to be able to read the status of the subchannels to all the receivers and it needs access to the status-signal to the arbiter. In section 4.4 it was stated that the status-signal may only have two drivers, one of which is the arbiter. From this it follows that requesting and releasing the access cannot be done by two separate processes, thus the HSM-sender process also has to request access to the channel.

The signal used to synchronise the sender task and the special HSM-sender process, called the HSM-status-signal, has the same type as the status-signal for mutual exclusion. In fact the sender task requests access to the channel indirectly via the HSM-sender process (see fig. 3.4). The HSM-sender process passes the status of the sender task on
to the arbiter and vice versa, except for releasing the channel, which will be delayed by
the HSM-sender until all receivers have accepted the data. Notice that because the
status-signal is used by the concurrent HSM-sender process that for each send statement
a status-signal has to be allocated. This in contrast with the mutual exclusion statements,
where one status-signal is used by all mutual exclusion statements (for the same object)
within one process.

The implementation of the send-statement for the handshake-merged protocol the sender
first requests access to the channel (via the HSM-sender process) and then waits for a
grant. Note that the grant also implies that the previous communication has finished else
the channel would not be free and thus the HSM-sender process could not return a new
grant. After receiving the grant, the sender can write the data to the channel. Since the
channel behaves like a register the sender can immediately disconnect its driver from
the channel. The sender also immediately indirectly 'releases' the channel again. In this
case this only tells the HSM-sender process that the data is now present on the channel.
While the sender-task now can proceed, the HSM-sender process sees to it that the rest
of the protocol is properly completed.

The HSM-sender process performs the following actions (see also fig. 3.5 and 3.4):
- wait till the sender request access to the channel
- pass on the request to the arbiter
- wait till the arbiter gives a grant
- pass on the grant to the sender
- wait till the sender releases channel (indicates that the data is on the channel)
- withdraw the grant
- inform all receivers that there is data available
- wait till all receivers have accepted data
- pass on release channel to arbiter

Note that assigning 'data_present' to the subchannels is also done by the HSM-sender
process. This means that the synchronisation between the sender and the receivers is
handled completely by the HSM-sender process. The advantage is that the subchannels
do not need to be connected to the sender process.

![Diagram](image)

**Fig. 3.4: The configuration implementing a Handshake Merged send-statement.**

Since none of the actions of the HSM-sender process accesses the channel, the HSM-
sender process implementation is independent of the data type of the channel. This
allows the HSM-sender process to be allocated in an entity whose component declar-
ation is added to the 'TL_TYPE' package. The problem that HSM-sender process has to be able to handle an arbitrary number of receivers, is solved by using a generic clause. The **generic clause** allows constants of the entity to be defined when the entity is instantiated (during VHDL compilation). Text 3.14 lists the declaration of the HSM-sender entity.

```vhdl
ENTITY hsm_sender IS
  GENERIC (n: POSITIVE);
  PORT (m8_channel : INOUT m8_status_type;
         me_ch_sender : INOUT me_status_type;
         recv_ev : INOUT event_type_arr (0 TO n-1));
END;
ARCHITECTURE hsm_sender_beh OF hsm_sender IS
BEGIN
  PROCESS
    VARIABLE ready: BOOLEAN;
    BEGIN
      IF m8_ch_sender/=request THEN
        WAIT UNTIL m8_ch_sender=request;
      END IF;
      me_ch_arbiter <= request;
      WAIT UNTIL me_ch_arbiter = grant;
      me_ch_sender <= grant;
      WAIT UNTIL me_ch_sender=free;
      me_ch_sender <= free;
      -- indicate 'data_present'
      FOR i IN recv_ev RANGE LOOP
        recv_ev <= (NOT recv_ev private, TRUE);
      END LOOP;
      -- wait until all receivers finished
      LOOP
        ready:= TRUE;
        FOR i IN recv_ev RANGE LOOP
          ready:= ready AND recv_ev[i].value;
        END LOOP;
        EXIT WHEN ready;
        WAIT ON recv_ev;
      END LOOP;
      me_ch_arbiter <= free;
      WAIT UNTIL me_ch_arbiter = free;
      END PROCESS;
    END;
END;
```

Text 3.14: The declaration of the entity for the HSM-sender process used to assist the send-statement of the handshake merged protocol.

Finally the implementation of the receiver. It turns out that the implementation of the receive-statement is almost the same as the implementation used for the none merged handshake. The only difference is that the message is now packed in a record with two fields as a result of the bus emulation. Text 3.15 lists the implementations of both the communication statements for the handshake merged protocol. Notice that (indirectly) requesting access to the channel and releasing it for the send statement is handled by a procedure call. These procedures are defined in 'TL_TYPES' package, this is possible
here because the procedures do not handle the channel and are thus data type independent. As a matter of fact the whole send statement could be implemented in a procedure, this is not done in order to be consistent with the implementation of the other protocols.

---

- in task 'tst1'
  - SEND msg1 TO tst2, tst3 ON chan;
  - wait_exccl_channel (tst1_me_ind);
  - chan <= (msg1, connected);
  - WAIT FOR 0 NS;
  - chan.id <= disconnected;
  - release_channel (tst1_me_ind);
  - WAIT FOR delay;

- in task 'tst2'
  - RECEIVE msg2 FROM tst1 ON chan;
  - IF NOT tst1_tst2.value THEN
    - WAIT UNTIL tst1_tst2.value;
  - END IF;
  - msg2 <= chan.data;
  - tst1_tst2 <= (tst1_tst2.private, FALSE);
  - WAIT FOR delay;

task1_sender: hsm_sender GENERIC MAP (2)
  PORT MAP (chan_me_ch (0), tst1_me_ind,
  recv_ev (0) => tst1_tst2,
  recv_ev (1) => tst1_tst3);

---

Text 3.15: Implementation communication statements for the handshake-merged protocol.

3.4.2.4 Rendezvous-merged protocol implementation

Like the handshake-merged protocol this protocol uses an arbiter to protect its channel and it uses the bus emulation for the channel. Only the channel does not behave like a register, thus the store is not used. There are status-signals needed, but only one for each process that includes a send-statement using the channel. So all send-statements using the same channel within one process, will use the same status-signal to communicate with the arbiter.

The implementation of the send-statement is almost the same as for the none merged rendezvous case. There are four changes to be made: One, the statement now has to request access to the channel, after it has made sure that all receivers can accept new data. Two, assigning the message to the channel has to deal with the bus emulation. Three, the driver has to be disconnected from the channel after the data has been accepted by all receivers. And four, access to the channel has to be released again. The implementation of the receive-statement has to be adjusted only to handle the bus emulation. Text 3.17 lists the implementations and text 3.16 lists the translated version of text 3.10.
ARCHITECTURE comm_arch OF comm_ent IS

COMPONENT comp1
    PORT (CHANNEL ch1 (tskA -> tskB);
        SIGNAL ch1: IN msg_type_ch_type;
        SIGNAL tskA_tskB: INOUT event_type);
END COMPONENT;

- PROTOCOL prot1: HANDSHAKE_MERGED;
- CHANNEL ch2;
    SIGNAL ch2: msg_type_ch_type;
    SIGNAL tsk2_tsk6: event_type;
- CHANNEL ch3:
    SIGNAL tsk3_tsk6: me_status_type;
    SIGNAL tsk4_me_ind: me_status_type;
    SIGNAL ch3_me_ch: me_status_type_arr (0 TO 1);
    SIGNAL ch3: msg_type_bus_type;
    SIGNAL tsk2_tsk5: event_type;
    SIGNAL tsk4_tsk1: event_type;
BEGIN
    inst1: comp1 PORT MAP (ch2 (tsk3->tsk6) ch2, tsk3_tsk6);

    proc1: PROCESS
        COM_DATA msg1: msg_type;
        VARIABLE msg1: msg_type;
        BEGIN
            tsk1: TASK
            BEGIN
                RECEIVE msg1 FROM tsk4 ON ch3;
                IF NOT tsk4_tsk1.value THEN
                    WAIT UNTIL tsk4_tsk1.value;
                END IF;
                msg1 := ch3.data;
                tsk4_tsk1 <= (tsk4_tsk1.private, FALSE);
                WAIT FOR delay;
            END TASK;

            tsk2: TASK
            BEGIN
                SEND msg1 TO tsk5 ON ch3;
                IF tsk3_tsk6.value THEN
                    WAIT UNTIL NOT (tsk3_tsk6.value);
                END IF;
                ch2 <= msg2;
                tsk3_tsk6 <= (NOT tsk3_tsk6.private, TRUE);
                WAIT FOR delay;
                msg2 <= exp1;
            END TASK;

            tsk5: TASK
            BEGIN
                RECEIVE msg3 FROM tsk2 ON ch3;
                IF NOT tsk2_tsk5.value THEN
                    WAIT UNTIL tsk2_tsk5.value;
                END IF;
                msg3 := ch3.data;
                tsk2_tsk5 <= (tsk2_tsk5.private, FALSE);
                WAIT FOR delay;
            END TASK;

            END PROCESS;

    proc2: PROCESS
        -- COM_DATA msg2: msg_type:= exp1;
        VARIABLE msg2: msg_type;
        -- COM_DATA msg3: msg_type;
        VARIABLE msg3: msg_type;
        BEGIN
            tsk3: TASK
            BEGIN
                SEND msg2 TO tsk6 ON ch2;
                IF tsk3_tsk6.value THEN
                    WAIT UNTIL NOT (tsk3_tsk6.value);
                END IF;
                ch2 <= msg2;
                tsk3_tsk6 <= (NOT tsk3_tsk6.private, TRUE);
                WAIT FOR delay;
                msg2 <= exp1;
            END TASK;

            tsk4: TASK
            BEGIN
                SEND msg2 TO tsk1 ON ch3;
                wait_exclude_channel (tsk4_me_ind);
                ch3 <= (msg2, connected);
                WAIT FOR 0 NS;
                ch3.id <= disconnected;
                release_channel (tsk4_me_ind);
                WAIT FOR delay;
            END TASK;

            END PROCESS;

    tsk2_sender: hsm_sender GENERIC MAP (1)
        PORT MAP (ch3_me_ch (0), tsk2_me_ind,
        recv_ev (0) => tsk2_tsk5);

    tsk4_sender: hsm_sender GENERIC MAP (1)
        PORT MAP (ch3_me_ch (1), tsk4_me_ind,
        recv_ev (0) => tsk4_tsk1);

    ch3_arbiter: arbiter(ch3_me_ch);
    ch3_store: ch3 <= (ch3.data, store);
END;

Text 3.16: VHDL translation of the text 3.10.
-- in task 'tsk1'
-- SEND msg1 TO tsk2, tsk3 ON chan;
IF tsk1_tsk2-false OR tsk1_tsk3-free THEN
  WAIT UNTIL (tsk1_tsk2-free AND tsk1_tsk3-free);
END IF;
wait_excl_channel (chan_me_ch(0));
chan <= (msg1, connected);
tsk1_tsk2 <= data_present;
tsk1_tsk3 <= data_present;
WAIT FOR 0 NS;
LOOP
  IF tsk1_tsk2-acknowledge THEN
    tsk1_tsk2 <= no_data;
  END IF;
  IF tsk1_tsk3-acknowledge THEN
    tsk1_tsk3 <= no_data;
  END IF;
  EXIT WHEN (tsk1_tsk2=data_present
             AND tsk1_tsk3=data_present);
  WAIT ON tsk1_tsk2, tsk1_tsk3;
END LOOP;
chan.id <= disconnected;
release_channel (chan_me_ch(0));
WAIT FOR delay;

-- in task 'tsk2'
-- RECEIVE msg2 FROM tsk1 ON chan;
IF tsk1_tsk2-data_present THEN
  WAIT UNTIL tsk1_tsk2-data_present;
END IF;
msg2 := chan.data;
tsk1_tsk2 <= acknowledge;
WAIT UNTIL tsk1_tsk2-no_data;
tsk1_tsk2 <= free;
WAIT FOR delay;

Text 3.17: Implementation communication statements for the rendezvous-merged protocol.

3.4.3 The 'special_types' package

All types used for communication are inserted in the 'special_types' package at the top of the VHDL file. The reason for this is that the type might not be visible at the point where the channel using it is declared. However in the VHDL translation this type is required for the channel declaration and thus has to be visible there. The problem is caused by the fact that the type of the channel is determined by the type of messages sent over the channel. A type declared between the channel declaration and the message declaration might be used for the message declaration. Text 3.18 shows an example of this problem. Note that only the procedures defined in 'coordinate_pack' can send or receive message of type 'coordinate_type'. A similar example can be made with two components (a sender and a receiver) using a type definition from a package which is made visible via a use clause in the process declaration part.

By copying the declarations of the types used for communication to the 'special_types' package, these types become visible throughout the design file (a use clause for this package is inserted before each primary unit). For a merged channel the used type has to be incorporated in the declarations for the bus emulation scheme, this is done in the same package. The type for a merged channel depends on the types of all the messages sent over the channel. According to semantics these types do not have to be exactly the same, it is sufficient that all types are subtypes (constraint versions) of one type. Therefore copying the (sub)type declaration of one such message object to the package and using that to declared the channel type is not correct, because there might be other messages sent over the channel that are less restricted.
This problem is solved by ignoring subtype declarations and thus any restriction to the type. Using the type declaration for the channel declaration ensures that all messages sent over the channel can be transported by it. A better solution would be to use the weakest constraint that can be found. Note that the latter needs quite some computations (all relevant type declarations have to be analyzed), so this is not done in the current limited compiler. See section 5.3.1 for more about this subject.

The type declarations which are required to declare channels, called target types, can use other type and subtype declarations, called intermediate types, e.g. to define the type of the fields of a record. Naturally the declarations of those intermediate types also have to be copied to the 'special_types' package. All this has to be done recursively because for the declaration of the intermediate types yet other type declarations might be used. By copying all these type and subtype declarations (from several declarative regions) to one package it is possible that one identifier is used for more than one type declaration. This of course is not allowed within one declarative region, so to avoid this problem all these declarations are numbered and the number is appended to the identifier.

In the VHDL file the original type declarations are replaced by a subtype declaration\(^1\) without a constraint, that makes the corresponding copied type declaration from the 'special_types' locally visible using the original type name. This is required because this type name is needed for the variable declaration that implements the com_data object.

\(^1\) Under certain conditions the VHDL simulator generates errors when this is used, however Model Technology Incorporated has acknowledge that this is a bug that will be fixed in the next release.
and it might also be used in other declarations. The implementation of communication constructs assigns the value from a channel to a com_data object and vice versa. VHDL strong typing demands that the same type declaration is used to declare the com_data and the channel implementation. Similarly the designer might have declared other objects that exchange values with the com_data object and thus must have the same type. The latter was already ensured in the correct TL design because TL also enforces strong typing, so by making the original type name usable again this correctness remains. Note that it is not necessary to replace the subtype declarations that are copied to the 'special_types' package, because VHDL considers subtypes to be of the same type anyhow.
4 TL TO VHDL COMPILER

4.1 Introduction

The job of the TL to VHDL compiler is to read a TL source file, translate the TL specific constructs and generate a VHDL file containing the implementation of the TL design in VHDL. In constructing the compiler two assumption are made:

- the TL code read by the compiler is assumed to be syntactically and semantically correct because the source text has passed the front end compiler that checks for syntactic and semantic errors
- for now it is assumed that the TL description is contained in one file

The parts of the source code that do not have to be translated (e.g. large parts of the procedural language) can be copied unchanged to the destination. Therefore the compiler reads the source file two times. During the first pass the parser builds the parser tree, during the second pass the code generator copies the contents of the source file to the destination. Things that have to be translated are changed into comment and the VHDL code implementing those things is inserted.

The Context Dependent Analyzer performs its task in between the above mentioned two passes over the source file. The CDA consists of three parts (see fig. 4.1). The first part traverses the parser tree according to the order of the contents of the source file. On the way an environment tree is built, data types for the 'special_types' package are added to a list (named type list) and some other details (discussed in the sections 4.3, 4.4 and 4.5) are computed. The second part traverses the data type list, while computing attributes of each element and adding more types to the list (see section 4.6).

The third part traverses the parser tree the second time, but now using a depth first search algorithm on the hierarchy of the TL design. The context information computed during this traversal (e.g. the number of mutual exclusion status-signals to be used) relies on the contents of the architecture and procedure bodies. Those bodies do not have to follow immediately their corresponding declaration, therefore following the textual order only ensures that the declaration is computed before the component or procedure is used, the corresponding body may not yet be computed and thus the desired information might not be available. On the other hand the first tree traversal cannot follow the hierarchical structure because at that time the location of the body corresponding to a declaration is not yet known.

For both the first and the third part of the CDA (and also the code generator) the order of traversing the parser tree and common computations (like computing the environment tree) are specified in one '.puma' file per tree traversal. The computations specific for translating a TL construct are specified in one file per TL construct per tree traversal.
4.2 Implementing an environment tree

The implementation of the environment is the backbone of any compiler as it gives access to the declarations of objects and types. The objective of the environment tree is to implement an efficient (i.e. fast) way to find these declarations. More precisely, the environment tree is used to avoid a linear search in the parser tree to find the searched declaration. The hard part of building an environment tree is ensuring that it agrees with the scope and visibility rules of the language. The scope and visibility rules for TL are the same as those for VHDL\(^1\) (see section 2.1.3) at least as far as they are applicable.

In order to incorporate the scope and visibility rules the declarations of the different declarative regions have to be separated. Moreover since a use clause only makes visible the declarations from a package declaration these should also be separated from the declarations in the package body. The environment has to permit that the declaration of an identifier is searched from the inner declarative region outwards (bottom up), this in contrast to the parser tree which only can (and has to) be traversed top down. Therefore the data structure to be used for the environment is a reversed tree (see fig. 4.2). To each node in the parser tree representing a declarative part, a pointer is set

\(^1\) Since the TL design is contained in one file (for now) the library clause is of no use yet in TL.
pointing to the corresponding environment node. This guarantees that during a tree traversal the correct environment node is always known.

USE WORKpak1.ALL;

ARCHITECTURE arch1 OF ent1 IS
  TYPE first ...
  CHANNEL second ...
  USE WORKpak2.ALL;
  ME_DATA third ...
BEGIN
  proc1: PROCESS
    USE WORKpak3.ALL;
    SUBTYPE fourth: ...
    COM_DATA fifth: ...
  BEGIN
  END PROCESS;
  proc2: PROCESS
...

Text 4.2: Example declarations.

Fig. 4.2: Environment tree of the example.

There is only one way to make a search for the declaration of an identifier efficient, namely sorting them in some manner. The direct consequence is that the textual order in which the identifiers were declared is lost. This is a problem because the textual order is important for the visibility of the declaration. Text 4.3 illustrates this with an example. Assume that all declarations from one declaration part (e.g. a package declaration, a process statement, a package body, etc.) are gathered and stored in one ordered data structure. So both 'tesevar' and the second 'enum' are placed together in one node of the environment tree and both have that node as their environment. Suppose that after the tree is built it is tried to locate the declaration of the type used to declare 'tesevar'. This search starts with the own environment and immediately finds the searched identifier 'enum'. However this is not the right one because according to the visibility rules that declaration cannot be 'seen' from the point where the variable was declared.

ARCHITECTURE beh OF system IS
  TYPE enum IS (first, declaration);
BEGIN
  PROCESS
    -- here only the above 'enum' type is visible
    VARIABLE test_var: enum;
    TYPE enum IS (second, declaration, hides, first);
...

Text 4.3: Example illustrating the importance of textual order.

Notice however that the environment tree is built by traversing the parser tree in the textual order. This means that when the node in parser tree representing the variable declaration is reached, the environment tree does not yet contain the second 'enum'
identifier. So when at that point the identifier 'enum' is searched then the right declaration will be found. This is the strategy used by the compiler to solve the problem described in the previous paragraph. Before the compiler can proceed to the next declaration, the found correct declaration has to be logged. This is necessary because else the correct declaration cannot be found again. For this purpose an attribute is added to the parser tree node representing the 'mark' (identifier 'enum' in the example) of the 'subtype indication' which defines the type of the variable. Here the mark has to refer either to a type or subtype declaration or to a standard type. When a mark is used in the statement part then it usually refers to an object or a subprogram.

For a correct implementation of the environment tree, the possibility of a declarative region consisting of two declarative parts (a subprogram or a design unit) has to be accounted for. Therefore the environment node has a pointer pointing to the environment node of the primary part, this pointer is only used for subprogram bodies and secondary units. When the environment tree is searched then the 'primary' environment node has to be searched before the surrounding environment of the secondary node is searched. After 'following' a pointer to the primary environment node, the surrounding environment of this primary node may not be searched. However this surrounding environment will be searched later anyway, because for units TL demands that the corresponding primary or secondary unit are located in the same library, this ensures that they both are surrounded by the same environment. For subprograms TL demands that the declaration and the body are declare in the same declarative region, which also ensures that they have the same declarative region as surrounding environment (although maybe not the same declarative part, e.g. for subprograms declared in a package).

The next problem is how to deal with use clauses. Use clauses can be found before and within a declaration part. The visibility rules state that the declarations from the package corresponding to the use clause only become indirectly visible when the same identifier was not already visible. That is, identifiers from a package can never hide identifiers declared in the current or surrounding declaration parts. This suggests that to find an identifier in the environment tree, first all environments corresponding to the current or a surrounding declarative region have to be searched. Only when this proves unsuccessful then the use clauses can be tried.

The compiler adds all use clauses found within or before a declarative part in a list that is attached to the corresponding environment node (see fig. 4.2). The use clauses are added when the corresponding node in the parser tree is reached, again this ensures that the use clause is not visible before this node is reached. Note that the order in which the use clauses appear in the list is not important. This comes forth from the rule that an identifier may not be made visible by two use clauses, not even if the use clauses are

---

1 In TL a subtype indication and mark are defined as:
   
   subtype_indication :::= mark [constraint]
   mark :::= identifier | selected_name
   
   Where selected_name may be used to refer to a not visible identifier (hidden or in a not visible package), however this is not yet implemented in the current compiler.
associated with another declarative part (a use clause cannot hide a declaration made visible by another use clause). This and relying on the fact that the design may be assumed to be correct, makes that the order of storing the use clauses is not important.

In searching the use clauses there is one more rule to be taken into account. This is the rule that a use clause can only make indirectly visible the declarations made in the package declaration. The declarations that were made visible in that package declaration by a use clause have to be excluded. Therefore a search may only test the identifiers that can be found in the ordered data structure of the environment tree identified by the use clause.

The following list summarizes a search in the environment tree, where of course the search ends when the searched identifier is found.

- search the current environment
- search the corresponding primary environment (if any)
- search the surrounding environment
- search the use clauses of the current environment
- search the use clauses of the corresponding primary node
- search the use clauses of the surrounding environment

To store the identifiers in an ordered data structure an unbalanced binary tree has been chosen. The reason for this choice is that the routines for building and searching such a binary tree are very simple. At some later stage another, more efficient, data structure might be implemented, which is quite easy because the data structure and the access to it are implemented following the rules of object oriented programming. The same is done for the rest of the environment tree. Text 4.2 shows an example with some declarations and figure 4.2 shows the corresponding environment tree.

4.3 Compiler for events

To translate the TL event constructs the CDA has very little to do. The computations needed can easily be done during the first tree traversal. The things to be translated are event declarations, event related statements, port maps of instantiated components and procedure calls. Only for the signal_event-statement are some computations required, all the rest can be translated by merely using the information that is present in the TL part to translate (and thus at the right place in the parser tree).

The signal_event-statement uses variables to keep track of which events have been signalled and which have not. These variables have to be declared in the declaration part of the process-statement or procedure body. The CDA collects for each process-statement or procedure body the names of all the events that occur in a list of a signal_event-statement within that statement part. For this purpose an empty list of identifiers is initialized when a node of a process-statement or procedure body in the parser tree is reached. The list is an attribute of the process-statement and procedure body node. When during the tree traversal the CDA reaches a node in the parser tree
with a signal_event-statement then the list of events (a child of that node) is traversed and every found identifier is added to the list.

For the port map of a component and procedure calls nothing has to be translated for the events in the interface. This is due to the fact that the event object is modelled by a single signal and that the identifier for that signal is the same as the identifier used for the event object.

### 4.4 Compiler for mutual exclusion

To translate the mutual exclusion constructs the CDA of the compiler has to compute several things, namely:
- for which (sub)types bus emulation and procedure declarations are needed
- the number of status-signals needed for an object
- the number of status-signals needed for a component instantiation
- the status-signals of the processes are gathered in an array, therefore it has to be computed which element of the array has to be used to translate the mutual exclusion statements

The declaration for the bus emulation and procedures are required for all the (sub)types used for a mutual exclusion object, which can be found by the me_data declarations. So when during the first tree traversal a me_data declaration is reached in the parser tree then an element is added to the global type list (unless it is already present) and a flag is set that indicates that the type is used for mutual exclusion. Like for the types used for communication (see section 3.4.3) these types and extra declarations are inserted in the 'special_types' package at the top of the VHDL file. This is only done to avoid that the bus emulation declaration are inserted twice for the same type (for both mutual exclusion and communication).

The declarations from the 'special_types' package are made visible throughout the VHDL design. This assures that the required declarations are visible where desired, i.e. at the point where the implementation of the mutual exclusion object is declared and where the mutual exclusion statements are located. Note that in the TL design the type used for the mutual exclusion object has to be visible at the point where the implementation of the mutual exclusion object is declared and where the mutual exclusion statements are located. Therefore inserting the bus emulation and procedure declarations directly after the used type declaration also would have been sufficient.

The compiler translates each mutual exclusion object to a signal modelling the data bus containing the actual object data, an array of status-signals which are eventually attached to the arbiter, and an alias declaration. The alias declaration makes that the original identifier of the mutual exclusion object refers to the field of the record that actually holds the value of the object. This trick makes that the references to the object
used in the statements don’t have to be changed by the compiler. For locally declared objects the compiler also generates an arbiter and a store. If the object is not used or only used by one process then the compiler generates a warning, because there is no need for mutual exclusion protection.

Before the compiler can translate the mutual exclusion objects it needs to know the number of status-signals required for each object. This number is the sum of the number of local processes using the object and the number of signals used by the components (instantiated entities). So to compute this number it is necessary to traverse the parser tree according to the hierarchical structure of the design (see section 4.1). Note that VHDL allocates one driver per signal for each process using that signal, even if there are more assignments to that signal in the process. This implies that if a process uses the same mutual exclusion object twice, it only needs one status-signal that can be used for both mutual exclusion access requests. Similarly a procedure never needs more than one status-signal per mutual exclusion object in its interface.

The processes and components using the mutual exclusion object need to know which status-signal(s) they should use, i.e. the compiler has to compute the number(s) of the status-signal array to be used. Since every status-signal is equal it is not significant which process gets which number. It is only important that the same element of the array is not used by another process. This computation can therefore easily be integrated with the computation of the number of status-signals to be used. The number(s) assigned to each use of the object status-signal(s) is registered in an attribute of the node in the parser tree representing the identifier of the object.

4.5 Compiler for communication

Due to the implicitness of the communication constructs, gathering all the information from the context is a considerable task. There are three simple things:

- protocol declarations, they are obsolete in VHDL and are thus simply reduced to comment.
- communication data objects declarations are translated by replacing the TL 'COM_DATA' keyword with the VHDL keyword 'VARIABLE'
- assignments to the communication data objects which are noted in TL using '=<'. Since message objects are translated to variables this has to be changed to ':=' (VHDL semantics). For this the compiler needs to check if the destination of a communication assignment ('=<') is such an object.

The following list lists the information that has to be collected by the CDA to translate the rest of the TL communication constructs:

---

1 In the current VHDL simulator using an alias in a procedure generates an error, however this is believed to be a bug.
For a channel object (includes subchannels)
- type of the channel
- direction of the message flow (for none merged interface channels only)
- names of implicit subchannels
- number of required mutual exclusion status-signals (only for merged channels)
- the required HSM-sender processes

For a communication statement
- protocol
- number of the mutual exclusion status-signal array element (only for send-statements in combination with merged protocols)

For interface associations of channels (component instantiations and procedure calls)
- numbers of mutual exclusion status-signals (only for merged channels)

For an architecture body
- implicit channels

For the whole design
- types used for none merged communication
- types used for merged communication

Although some information is not always needed, it is still always computed. This has two advantages. One, the compiler can handle communication constructs uniformly without the need to check the protocol type constantly. Two, the additional information can be used to check if it contains the only allowed value. If the check fails then a warning can be given.

4.5.1 Setting channel types

Before the signal that implements a channel can be declared in VHDL its type has to be known. This type depends on the messages sent over the channel and thus can be determined when the channel is used in a communication statement. The channel type might also be determined when the channel is connected to another channel in an interface association. In the latter case the types in the instantiated component or the called procedure must have been set before, i.e. the body of the item must have been processed; therefore this can only be done during the second tree traversal. On the other hand registering the types when it is used in a communication statements is done during the first tree traversal.

Note that TL semantics allows that a channel is not used by any communication statement. In such a case the type of the channel will not be known and thus a signal implementing the channel cannot be declared. Instead a warning will be generated, and the (interface) channel declaration and the related interface associations will be put into comment.
A type used for communication is added to the global type list (unless it is already present). This is done during the first tree traversal at the same time the type of the channel is set. For each type added to the list it has to be known what extra declarations have to be generated for it in the 'special_types' package. Therefore a flag is set that indicates if the type is used for merged or none merged communication. After the first tree traversal all communication types are known, because a type can only be used for communication via a communication statement. The intermediate types that also have to be inserted in the 'special_types' package are gathered by the second part of the CDA, which is discussed in section 4.6.

4.5.2 Gathering implicit subchannels

In order to register the implicit subchannels related to a channel, it is sufficient to record the subchannels used by a send-statement or an interface association (not a receive-statement). This can be seen if one observes that for a subchannel used by the receive-statement, in a correct design, three situations can be recognized. The first two possible situations are that the subchannel is used by a local send-statement and in an interface association, in which case the subchannel will be recorded when that partner is visited. The third possible situation is that the subchannel is connected to the outside world of the entity, which means that the subchannel has to be explicitly declared, so there is no need to record the subchannel.

Before a subchannel is registered, originating from a send-statement or an interface association, it has to be made sure that the subchannel is not already declared. The subchannel may already be declared when it is used by both a send-statement and in an interface association or it may be explicitly declared in the interface. All implicit subchannels have to be declared in the declaration part of the architecture, while the explicit subchannels are declared by replacing their TL subchannel declaration in the interface; therefore the recorded implicit subchannels are marked.

4.5.3 Setting the channel direction

The channel direction is set implicitly by setting the direction of all its subchannels. The direction of the channel is only required for a none merged channel, in which case all subchannels should have the same direction and thus the direction of the channel can be determined by reading the direction of its first subchannel. For merged channels the channel is implemented by a bus and the mode is always INOUT.

In the current compiler the direction of the subchannels is not used because all subchannels have two drivers and thus must have mode INOUT. However it is expected that this attribute will be used when the TL protocol specification is incorporated in the compiler.

The direction of a subchannel is set when the subchannel is registered (or checked to already exist) originating from a send-statement. This cannot yet be done for
subchannels registered or checked coming from an interface association because the direction of the attached subchannel might not yet have been computed. Therefore copying the direction is postponed till the second time the interface association node is visited (at the same time the channel type is registered). From the above it follows that the direction is never set for explicit subchannels used by a receive-statement. This is obviated by setting the direction of a subchannel default to IN.

**4.5.4 Computing mutual exclusion numbers for communication**

The computation of the numbers for mutual exclusion channels is virtually the same as for the translation of mutual exclusion objects. In the case of the handshake-merged protocol each send-statement (HSM-sender process) must get its own status-signal, whereas for the rendezvous protocol one status-signal per process suffices. Regard that an entity or procedure that only encloses receive-statements for a interface channel does not need status-signals to be declared in its interface. This fact can simply be detected by checking the number of status-signals reserved for the senders in the entity or procedure, which in this case will be zero.

The number of senders on a channel is always computed irrespective of the channel protocol. This enables the compiler to generate some warnings. A warning is generated when a merged protocol is used for a channel with only one or no sender and conversely, when for a channel with more than one sender a none merged protocol is used.

**4.5.5 Computing the required HSM-sender processes**

For each send-statement with the handshake merged protocol a HSM-sender process has to be added. When the send-statement occurs within the statement part of a process-statement then the HSM-sender process can be allocated in the same architecture body. However if the send-statement occurs in a procedure then this HSM-sender has to be allocated outside the procedure (concurrency is not possible within a procedure) and therefore the need for such processes has to be indicated in its interface. That is the reason why the list of HSM-senders is added to a channel, which in the case of a procedure must be located in the interface.

For the handshake merged protocol the mutual exclusion status-signal is only used by the HSM-sender process so this signal can be removed from the procedure interface. Similarly the synchronisation over the subchannels is handled by the HSM-sender process, thus they can also be removed from the procedure interface. Instead a HSM-status-signal has to be added to the interface for each send-statement, this signal is emulated by the first subchannel named in the send-statement. To indicate these special situations a flag is added to each subchannel, this can have one of three values:
- special: the subchannel is used to emulate the HSM-status-signal
- not_used: the subchannel is not used
- normal: the subchannel is used normally (e.g. for other protocols)
When during the first tree traversal a send-statement for the handshake-merged protocol is encountered then an element is added to the HSM-sender list. If the send-statement is located in a procedure then all related subchannels are marked 'not_used', except the first one that is marked 'special'. If the send-statement is located in a process-statement then the subchannel flag is set to 'normal', this is also the default value. During the second tree traversal the information of interface associations is copied. Here three different situations have to be recognized:
- a procedure call within a procedure
- a procedure call within a process statement
- a component instantiation

In the first case both the HSM-sender list and the flags of the subchannels have to be copied to the corresponding items in the interface of the outer procedure. In the second case only the HSM-sender list may be copied. The subchannels flags may not be copied, because now even the 'not_used' or 'special' used subchannels are needed to be connected to the HSM-sender which will be located in the current architecture body. In the third case all information on HSM-senders has to be ignored.

4.5.6 Computing other communication context information

When during the first tree traversal the CDA comes across a send-statement without a channel specification an implicit channel declaration is made. The implicit channels are gathered in a list added to the node in the parser tree representing the architecture body of the entity (within procedures implicit channels are not allowed). This is done here because the declarations to be generated for these channels have to be appended to the declaration part of the architecture body. Since the implicit channels can only be used locally there is no need to register implicit channels when the counterpart receive-statement is encountered by the compiler. However a reference has to be set from this receive-statement to the implicit channel declared 'by' the send-statement. This can only be done during the second tree traversal because there is no rule that says that a send-statement using an implicit channel has to be used before the corresponding receive-statement appears.

The protocol type used for an implicit channel is the so-called implicit protocol. The default protocol used for channels not specifying a protocol is called the default protocol. Both protocols default to the handshake protocol, but this can be changed via a command line option of the compiler. Note however that it is not very wise to use a merged protocol for the implicit channels because these channels can never be merged.

When the compiler meets a communication statement during the first walk through the tree then the protocol used by the related channel is copied to an attribute with that statement. Due to the same reason as before this cannot yet be done for the receive-statement without a channel specification, this action is therefore also delayed.
4.6 Gathering intermediate types for 'special_types'

During the first tree traversal a list is made of all the data types used for mutual exclusion and communication (the target types). The second part of the CDA traverses this list and gathers all intermediate types (types used in the declaration of target types or other intermediate types). All these declarations are eventually inserted in the 'special_types' package at the top of the VHDL file. The order in which they are inserted is equal to their textual position in the TL source file. This guarantees that all declarations that are used within the package for other declarations are visible to the latter.

The order of the declarations in the package is achieved by reverse ordering the elements of the list according to their textual position, which also helps in preventing a type to be added twice to the list. The reverse order is used because gathering the intermediate types is done in the reverse textual order. Since types always must be declared before they are used, using the reverse textual order ensures that found intermediate types always have to be added after the type list element that is being processed. This scheme guarantees that when the end of the list is reached that all elements of the list have been processed and that all required declarations are present.
5 EXTENSIONS TO THE COMPILER

This chapter contains ideas to extend or optimize the current implementation of the VHDL constructs or the compiler. None of these ideas have been implemented yet.

5.1 Extensions to the event implementation

In the current compiler the code implementing the event statements is inserted in-line because general procedures that can be used for all possibilities allowed by the TL semantics cannot be made. However it is assumed that the majority of the event statements only have one event as parameter. By making procedures that implement this special case is quite easy and it will noticeably simplify the VHDL code if the assumption is correct.

The effort to implement this optimization in the compiler is small. All that has to be done is adding the procedures to the 'TL_TYPES' package, and in the CDA and the code generator an exception has to be made for event statements with one event. The latter can be done using a rule spanning several tree nodes.

Secondly an extension could be made to the event construct in TL allowing the event to be initialized to true. Currently this is not allowed by the TL language definition, but the VHDL implementation can handle it. To implement this, the value field of the signal modelling the event object should be TRUE, the value of the private field is not relevant. The signal declaration to be initialized is the signal declared in the declaration part of the architecture body. The syntax of TL (and also VHDL) assures that there is always only one such declaration, where all other declarations (if any) are port declarations.

The third and last idea to extend the TL event construct in TL is to allow events to be declared implicitly. This can only be allowed for events that are used locally, i.e. the signal_event and the wait_event-statements for the event occur in the same entity. The way this could be handled by the compiler is similar to the way the implicit subchannels for communication are handled (see subsection 4.5.2).

5.2 Extensions to the mutual exclusion implementation

The next subsections contain three alternative ideas for mutual exclusion algorithms and their implementation.
5.2.1 Hierarchical arbiter

The currently implemented arbiter is a global arbiter, as there is only one (per mutual exclusion object) for the whole design. As an alternative there could be an arbiter allocated for each entity, this implementation is called the hierarchical arbiter. For the hierarchical arbiter implementation there is one master arbiter at the highest hierarchical level at which the object appears. This master arbiter can get requests from local processes (i.e. in same entity) or from slave arbiters located in components (see fig. 5.1). The slave arbiters also can get request from local processes or other slave arbiters in components. When these slave arbiters have received a request, they have to ask the arbiter one level higher in the hierarchy (their master) for permission to hand out a grant.

![Diagram of hierarchical arbiter](image)

Fig. 5.1: The hierarchical arbiter.

When a slave arbiter has received permission to hand out a grant, it has to allow all its requesters to gain access once and then it has to return the grant. The requesters may be scanned only once because if the slave arbiter postpones giving up the permission to hand out grants until none of its requesters desires access to the object then this could starve the processes not seen by the slave arbiter (i.e. a process on a higher level or in another branch of the structural hierarchy). On the other hand the slave may not return the permission to hand a grant as soon as its requester has finished, because that would favour processes that directly put their request to the master arbiter over the processes somewhere deep down in the hierarchy.

In the section 4.4 it was described how and why the number of mutual exclusion status-signals has to be computed. Evidently the number of status-signals needed in the entity interface depends on the contents of the body of the architecture! This is a problem as it violates the principles of hierarchy (the entity declaration should contain all information needed to use it as a component). This especially becomes a problem when an entity uses itself recursively¹. With the current implementation (one arbiter) the compiler has to completely unfold the recursion to find out the total number of status-signals to be used for such an entity. To be able to do this, the compiler has to incorporate an

¹ This is allowed when the GENERATE statement is used in combination with the GENERIC port clause.
interpreter for a small part of the TL language. Using the hierarchical arbiter prevents all these problems because every entity using the object always uses only one status-signal, clearly this will considerable ease the job of the compiler.

However there are also two disadvantages for the hierarchical arbiter implementation; first of all the number of arbiters is quite large, which will claim a lot of the available resources (chip area or memory space). On the other hand for a hardware implementation this idea might be used to reduce a sizable bundle of status-signals running across the IC surface to one status-signal. Secondly a request for access from a process has to be passed upwards through several slave arbiters and that will be payed for by time. These disadvantages can be reduced somewhat by at least removing the slave arbiters that serve only one requester.

5.2.2 Token-ring mutual exclusion algorithm

Next to the mutual exclusion algorithm with an arbiter there is one other algorithm that can be used to implement mutual exclusion. This is an algorithm based upon the token-ring idea. The token-ring algorithm puts all processes using the mutual exclusion object in one large ring (see fig. 5.2). One of those processes has the token, which means that it has the permission to access the object. However this process might not want to access the object at that moment, in that case the process has to pass the token to its neighbour on the ring. This way the token gets passed around the ring until it ends up in a process that wants to access the object. In that case the process holds on to the token until it is finished with the object.

![Fig. 5.2: A token-ring.](image)

Obviously the processes in the ring can do little else than handling the token; therefore this has to be a dedicated process that is added to each TL process that uses the mutual exclusion object. The TL process passes its access request to its dedicated process and the dedicated process grants the access as soon as it gets hold of the token. When the TL process is finished, it reports this to dedicated process, which then passes on the token.

---

1 A token-ring is one of the many possible networks to interconnect a set of computers.
The advantages of this algorithm are:
- There is no need for one 'complex' arbiter that does the global bookkeeping for the mutual exclusion object.
- Like the hierarchical arbiter this algorithm has the quality that it uses a fixed number of signals for each mutual exclusion object in the interface.

The disadvantages of this algorithm are:
- For each process accessing the object a dedicated process has to be added.
- If there is no process requesting access to the object then the dedicated processes are very active in passing around the token.
- The time for the token to get passed around the ring slows down the system. This might be a considerable time for large rings.

From the above two lists it can be concluded that the token-ring will hardly be usable for a software implementation because it consumes too much processor time with doing nothing, that is passing around the token. On the other hand this is not a problem when the object is heavily used and thus the token is held by a dedicated process most of the time.

Further it can be inferred that the token-ring versus the arbiter trades one large controller (the arbiter) for a number of small controllers (the dedicated processes). It is estimated that in hardware the advantages of a small the token-ring will outweigh those of the currently used arbiter.

**5.2.3 Mutual exclusion with priorities**

The mutual exclusion algorithms described so far assume that every requesting process is equal to the other. However sometimes it is desirable to give some mutual exclusion requests priority over the rest. A mutual exclusion algorithm with priorities can be implemented in several ways. A simple and correct solution, is to allocate one arbiter for each used priority level. The arbiter handling the highest priority request is the master arbiter and is almost equal to the currently implemented arbiter. The only difference occurs in the situation that the arbiter has not received requests from one of its processes. In that case the arbiter passes the 'token' on to the arbiter that handles the requests of one priority level lower, at least if that arbiter is requesting the token.

The slave arbiters behave almost equal to the master arbiter, except that they first will have to apply for and receive the token before they can hand it out to any of its processes or the next slave arbiter requesting the token. Note that while the slave arbiter is waiting for the token that was requested by the next slave arbiter one of its own processes might also have generated a request. Therefore the slave arbiter always needs to check the status of its processes after it has received the token and before it passes it on to the next arbiter. When the token is returned to a slave arbiter it always has to return it immediately to its master, else a process with a higher priority that has requested the token gets shut out while the slave arbiter generously passes the token to one of its processes.
A bit more complex algorithm to implement mutual exclusion uses only one arbiter. This arbiter receives all requests for the object via several arrays, one for each priority level. The arbiter has to search for a requesting process starting by scanning the array of the highest priority processes and gradually descending to the lower priorities until a request is found. To ensure fairness the arbiter has to log for each array (priority) which element (process) last used the object, so that the arbiter knows where a scan of the array has to start.

Another simple solution would be to use one arbiter with all the status-signals in one array and let the status-signal not only carry the request but also the priority of the request. The arbiter would search the array of status-signals for the request with the highest priority. However this is a faulty implementation as it does not treat two processes with an equal priority fairly. Imagine for example that there are three processes named A, B and C whose status-signals are mapped to the array in alphabetical order. Further assume that process A and C have a low priority whereas process B has a high priority. If process B and C occupy the object all the time then process A will soon starve (see fig. 5.3). This is caused by the fact that the arbiter always scans the array starting with the successor of the process that just finished. The latter is required to ensure fairness among the processes of the highest priority.

Note that there is a significant difference between this last inaccurate implementation and the two first implementations. The difference is that the last idea allows the process to determine dynamically the priority of the request, whereas the two other implementation enforce that the priority is established during compilation. It is probably possible to improve the algorithm used by the arbiter of the last implementation so that the arbiter will be fair, however this will cost quite some administration and thus resources. It is estimated that there is little use for a mutual exclusion algorithm with dynamic priorities and thus this idea is not further researched.

For the implementation of a mutual exclusion algorithm with fixed priorities the priority of a request has to be added somewhere to the TL language. The most logical place seems to be to add the priority (a constant) to the wait_exclusion-statement. Another possibility is to add the priority to the declaration of the object. However this would imply that all local processes always have the same priority and that unnecessarily limits the power of priority extension.
5.3 Extensions to the communication implementation

5.3.1 Optimizing communication types

In section 3.4.3 it was stated that it would be better to use the weakest constraint for the declaration of the type to be used for the channel declaration. However even if the weakest constraint would be used resources could be wasted for the communication implementation. Assume for example that one type declaration is used for two independent channels named A and B, of which B is more constrained than A. Using the weakest constraint of those two channels would allocate too much resources for channel B.

For merged channels a better solution might even require type conversion functions. Assume for example that to each channel in the above example a different instantiation of the same entity is connected. This entity only uses the more constrained version of the type. Since it has to be connected to both the channels VHDL strong typing demands that either both channels have the same type or a type conversion function is used when the entity is connected to the less constrained channel. Note that the problem is caused by the declarations needed for the bus emulation. These declarations use the channel type for one field of a record and VHDL does not allow the type of a record field to be constrained after the record declaration.

For the none merged channels there are no problems because TL semantics demands that the subtype of the message object used by the sender and receiver are exactly the same. This implies that when this subtype is used to declare the none merged channel then no resources are wasted.

5.3.2 Buffers for communication

For some communication actions one of the communicating partners exhibits strong peaks in the desire to communicate (e.g. as a result of time slicing in a software environment). If the counterpart also has such strong peaks but at different times or it can only handle a slow yet steady stream of messages, then the net effect will be that very little traffic occurs between the two partners. In such a case the designer might decide to intercept message peaks by adding a buffer between the two poorly communicating tasks. In the sequel a process that exhibits peaks in its desire to communicate is called a peak load process.

From the start it has been the intention to assimilate buffers into the communicating constructs offered by TL. First however there is a fundamental question to be solved: where? Especially in combination with merged channels the answer to this question can
have a significant influence on the expected increase in performance. For the moment four different possibilities are recognized:

- between the sender and the channel (before the channel, see fig. 5.4(a))
- between the channel and the receiver (after the channel, see fig. 5.4(b))
- split in two parts, one part before and one part after the channel (see fig. 5.4(c))
- as a part of the channel itself (see fig. 5.4(d))

For none merged channels all four possibilities will have the same effect, so in the sequel it is assumed that communication to be buffered uses a merged channel. This implies that apart from the poor communication between the concerned partners the throughput is also disturbed by actions of the other users of the channel.

For example, when two peak load processes finally get synchronized and both want to transfer a set of messages then this will be frustrated by any other communication pair that decides to use the channel. Note that this is caused by the fairness of the mutual exclusion algorithm. This observation suggests that giving the two peak load processes the highest priority also could solve the problem. Another (simpler) possibility is for the sender to cheat by holding on to the exclusive access of the channel till the whole set of messages has been transferred.

A better solution will be to add a buffer on each side of the channel. Provided that the buffers are large enough, this not only permits the two tasks to perform their communications at their own pace, but it also removes the need to synchronize them. When only one of the two communicating processes is a peak load process and the other process wants to communicate slow an regularly then only a buffer on the side of the peak load process will probably be sufficient to streamline the communication.

Now imagine another situation: assume that there are two tasks that are most of the time busy with computations and on a regular bases a message is passed from one to the other. Normally there is no need for buffering in this situation. However if a set of tasks for a short time uses the same channel to the limit of its capacity, then the two tasks would hardly be able to communicate. Consequently both tasks would cease working, in order to wait for their communication. Here a buffer on each side of the channel enables the sender to continue working while putting the messages in the

Fig. 5.4: Possible positions for a buffer.
buffer. When the channel is no longer used by the set of heavily communicating tasks, it can be used to transfer the contents of one buffer to the other buffer. This would allow both tasks to continue working when the set of tasks again starts to use the channel heavily. When one of the two tasks is primarily concerned with passing on the messages then that task would have no use for a buffer, as it can process the messages faster than its peer anyway.

By the way, note that buffering the receiver(s) has one important advantage over buffering the sender. This comes forth from the fact that the channel is claimed by the sender and the sender has to hold on to the channel till all receivers have accepted the message. This mandatory wait (if there is at least one slow receiver) reduces the throughput of the channel significantly if it is intensely used. Adding a buffer on the sender side does not change this. Conversely adding large enough buffers before the receivers makes that each message can be transferred swiftly, which will profoundly increase the efficiency of the channel. However this only works when the sender is not inherently faster than the slowest receiver, because in that case a buffer of any size will eventually be full and the sender holds on to the channel while it waits till one place becomes available in the buffer.

Up until now all efforts were directed at avoiding that the communication tasks had to wait, there is however also the expense of that strategy to be considered. For example, in a situation where a send-statement services several receivers it will clearly be more costly to add a buffer before each receiver than only one on the side of the sender. Also splitting a buffer into two halves is likely to increase the requirements, because each buffer will need its own controller.

There is still one buffer position not discussed at all, namely the buffer that is a part of the channel. This buffer buffers all communication actions over the channel. Here a sender passes its message to the buffer and it tells to which receivers the message is addressed. Subsequently the buffer will take care that the message is passed on to the addressees. As a matter of fact this buffer behaves like a small Packet Switch Exchange (PSE) in a Packet Switched Network (PSN).

The 'buffer' can be made to be very flexible. For example, it might accept a message for some receivers which it subsequently passes on to each of the receivers separately. This means that the single communication action is split up into several, which means that the receivers associated with one sender do not have to accept the message consecutively. Meanwhile the 'buffer' could allow a second message to overtake the first. Obviously expanding the capabilities of this buffer has to be paid for by resources. Note that it could be useful to combine this buffer with the previously discussed private buffers for the senders and receivers.

From the above discussion it should be clear that there is not one optimal solution for all situations. The optimal solution has to be judged for every individual case based upon the behaviour of all the senders and receivers using the channel and of course the available resources.
There are two practical structures for a buffer recognized (which will hopefully soon be made available in the TL to VHDL compiler). The first is a ring buffer and the second is a large shift-register. Both use the well known FIFO scheme (First In First Out). When the ring buffer is used for a none merged channel with one receiver, already implemented in VHDL by M. Boersma [Boel], then there is no extra process needed. Both the sender and the receiver operate on their own pointer to an element of a circular array. The pointers indicate where the next message respectively should be written and read.

If the buffer has to be used with more receivers or in combination with a merged channel then the buffer has to be made autonomous, i.e. a concurrently running buffer process has to be allocated. Now the buffer also has to incorporate a receiver and a sender-task (not necessarily concurrent) to communicate with the original receiver and sender.

The shift-register is always autonomous, in fact every element of the shift-register is autonomous. Each such element tries to receive a message when it is empty and it tries to pass on the message when it has one. This means that here each element incorporates a receiver and a sender-task (always sequentially).

It is estimated that for a hardware implementation at least for small buffers the shift-register uses less hardware compared to the ring buffer. This comes in handy when a small buffer has to be added to both sides of the channel. On the other hand it takes time for a message to propagate through the buffer. So it seems that the using the shift-register instead of the ring buffer exchanges hardware for time. For a software implementation the shift-register has no right to exist. This follows from the fact that it keeps moving data and for that it uses a concurrent process for each element of the buffer (task switching).

Both buffers must of course adapt their interface protocols to the protocols of the tasks to which they communicate. If the buffer is used on the side of the sender then the synchronization with the receivers naturally has to be added at the end of the buffer. The latter is very similar to the functionality of the HSM-sender process. The buffers can also be deployed between the sender and a merged channel. For such a situation the buffers should also be equipped with a mutual exclusion status-signal to accommodate communication with the arbiter. The buffers inevitably have to be type dependent, so at best one entity declaration to implement a buffer per data type is sufficient.

5.3.3 Protocols with receiver initiative

In the two currently used protocols (handshake and rendezvous) the initiative for a communication action is taken by the sender. Assuming that a previous communication action has been properly finished, this means that the send statement offers the data to the receivers irrespective if they are ready to accept new data (see fig. 3.3). In a situation where the receiver takes initiative the sender would have to wait until the receiver indicates that it is ready to accept data before it could sent the data (see fig. 5.5).
Fig. 5.5: The rendezvous protocol with receiver initiative.

Note however that leaving the initiative to the receiver is prohibited by the semantics of TL when a sender sends the message to more than one receiver. Because in that situation the sender can only send its data when all receivers indicate that they are ready to accept a message. This means that all the ready receivers have to wait if there is one receiver that is not ready and this is forbidden by the TL semantics.

The advantage of leaving the initiative to the receiver is that the channel is only shortly used. This is because the channel is only used when the sender and all the receivers are ready to transfer the data and finish the communication action. This in contrast with the situation where the sender has the initiative. Here the channel is occupied while it might take one of the receivers a long time before it is ready to accept new data. From this it follows that giving the initiative to the receiver can significantly improve the throughput of a merged channel. Note that this scheme, as opposed to buffering, even works when the sender is inherently faster than the slowest receiver.

Giving the initiative to the receiver has one major disadvantage and that is that it can cause deadlock. This possibility is illustrated by text 5.1, here process 'proc2' first waits for an event before it initiates receiving data. However this event will never be signalled because the signalling process 'proc3' only signals this event after it has received its data and this data never arrives because the sender has to wait until 'proc2' has reached the receive statement. When the sender has the initiative then this deadlock would not be present because then 'proc3' can receive its data independent of 'proc2'.

Giving the initiative to the receiver can easily be emulated even with the current limited compiler. To do this a signal_event-statement has to added just before each receive statement and just before the corresponding send statement a wait_event-statement that waits for all related events has to be added. As a matter of fact giving the initiative to the sender or receiver is an aspect of the protocol between the communicating partners. The protocol declaration was added to the TL language to separate the specification of the protocol from the other computations.
Text 5.1: Deadlock caused by giving the receiver the initiative.
CONCLUSIONS

In this report it was shown how a part of a TL to VHDL compiler was constructed. The current compiler can handle most of the TL language that does not have to be translated and it translates the TL event, mutual exclusion and communication constructs. The latter construct currently lacks buffers and protocol specifications. When the current compiler is compared with the previous compiler [Boe1] it can be noted that it:
- is build using the GMD-toolbox, which makes it much easier to maintain
- is structured so that it easily can be expanded
- incorporates an environment tree obeying the scope and visibility rules
- can interpret the most commonly used use clause
- can translate all three TL constructs used in a hierarchical design
- can translate procedures
- improves the implementation of the event object (only one signal instead of two)
- improves the implementation of the mutual exclusion object by using data bus emulation to model the mutual exclusion object
- improves the implementation of the mutual exclusion statements by using procedures
- can also use the rendezvous protocol to translate the communication constructs
- can translate communication over merged channels
- solves the problem of the visibility of the type declarations needed for the channel declaration

Alas the current implementation of the mutual exclusion algorithm violates the principles of hierarchy by making the contents of the entity interface dependent on the contents of the corresponding architecture body. This is especially a problem for translating systems that use recursion. Two of the three alternative implementations for the mutual exclusion algorithm, do not violate the principles of hierarchy. These two algorithms are the hierarchical arbiter and the token-ring. Using the token-ring instead of the arbiter offers a way to exchange hardware for time, at least if the mutual exclusion object is shared by a small number of processes. The third alternative mutual exclusion algorithm extends mutual exclusion with priorities and thus can be used to make the TL mutual exclusion construct more powerful.

The current implementation of a communication channel may waste resources, this is a result of the implicitness of the TL language. However improving this proves to be a difficult task.

Communication can benefit from buffering in several ways:
- it removes the need to synchronise the sender and the receiver, which reduces the time that those tasks have to wait for each other
- it can prevent a communicating task from having to wait for a channel that is being used by others
- it can reduce the time a communication action occupies the channel, which increases the through put of the channel
- it can intercept and smooth peaks in a stream of messages, which enables each task to communicate with its own pace and increases the through put of the channel
Taken together it can be concluded that the behaviour of the sender and the receiver, and the utilisation of the channel determine the optimal place and size of the buffers.

Two possible implementations for a buffer have been compared, a ring-buffer and a large shift-register. Due to the many concurrent parts that constitute the shift-register it is not suitable for a software implementation. If the size of the buffer is small then using the shift-register instead of a ring-buffer exchanges time for resources.

The semantics of TL communication demands that a sender takes the initiative in a communication action. When instead the initiative would be given to the receiver then this could increase the throughput of the channel, however this also easily can cause deadlock.

Before the TL to VHDL compiler is elaborated with the above mentioned ideas for alternative implementations and extensions, it first has to be made complete. The things that still have to be added to the compiler are:
- selected names and related to that less common use clauses
- request and acknowledge synchronisation
- protocol specification (includes buffers)
- communication if, case and loop statement
- shared processing
- parallel tasks
- interrupt tasks
- time out conditions

Only when all these things can be translated by the compiler the power of the TL language becomes available for practical designs.
BIBLIOGRAPHY


[Dij] Dijkstra E.W.; "The superfluity of the general semaphore"; internal paper Eindhoven University of Technology, TUE EWD 734;


Appendix A: 'TL_TYPES' library file

The following pages list the TL library file that should be located in the 'TLVHDL' library.

Note that in order to include the declaration of the HSM-sender component in the 'TL_TYPES' package its declaration has to precede this package. However the HSM-sender uses some of the 'TL types' so these have to be declared before the HSM-sender. This problem is solved by using two packages. The first package only defines the 'TL types' that are needed for the HSM-sender component. Here the identifiers that have to be made available in the real 'TL_TYPES' package are prefixed with 'pre_'. These types are made available in the second package, using the identifier without the 'pre_' prefix, by using a subtype declaration without constraints.
PACKAGE tl_pre_types IS

-- EVENTS

TYPE event_record IS
  RECORD
    private : BOOLEAN;
    value : BOOLEAN;
  END RECORD;

TYPE event_record_array IS ARRAY (NATURAL RANGE <>) OF event_record;

FUNCTION resolve_event_record (drivers : IN event_record_array) RETURN event_record;

SUBTYPE pre_event_type IS resolve_event_record event_record;

TYPE pre_event_type_arr IS ARRAY (NATURAL RANGE <>) OF pre_event_type;

FUNCTION resolve_event_record (drivers : IN event_record_array) RETURN event_record IS
  VARIABLE sig : NATURAL;
  BEGIN
    IF drivers'LENGTH=2 THEN
      -- identify driver
      IF drivers(0).value THEN
        -- driver 0 is the signal statement
        sig := 0;
      ELSE
        -- driver 1 is the signal statement
        sig := 1;
      END IF;
      -- determine event value
      IF drivers(0).private XOR drivers(1).private THEN
        -- event is just made true, return signal statement value
        RETURN drivers(sig);
      ELSE
        -- event is just made false, return wait statement value
        RETURN drivers(1-sig);
      END IF;
    END IF;
    RETURN drivers(drivers'LEFT);
  END;

FUNCTION resolve_me_status (driver : IN me_status_array) RETURN me_status IS
  BEGIN
    -- signal can only have one or two drivers
    IF driver'LENGTH = 1 THEN RETURN driver(driver'LEFT); ELSEIF driver'LENGTH = 2 THEN
      IF ((driver(0)=request) OR (driver(1)=request)) THEN
        IF ((driver(0)=grant) OR (driver(1)=grant)) THEN
          RETURN grant;
        ELSE RETURN request;
      END IF;
      ELSE RETURN free;
    END IF;
    ELSE RETURN free;
    END IF;
    -- other cases (including errors)
    RETURN free;
  END;
END;
END;

PACKAGE BODY tl_pre_types IS

-- EVENTS

FUNCTION resolve_event_record (drivers : IN event_record_array) RETURN event_record IS
  VARIABLE sig : NATURAL;
  BEGIN
    IF drivers'LENGTH=2 THEN
      IF drivers(0).value THEN
        sig := 0;
      ELSE sig := 1;
      END IF;
      IF sig THEN RETURN drivers(0);
      ELSE RETURN drivers(1);
      END IF;
    ELSE RETURN sig;
    END IF;
  END;

FUNCTION resolve_me_status (driver : IN me_status_array) RETURN me_status IS
  BEGIN
    IF driver'LENGTH = 1 THEN RETURN driver(driver'LEFT); ELSEIF driver'LENGTH = 2 THEN
      IF ((driver(0)=request) OR (driver(1)=request)) THEN
        IF ((driver(0)=grant) OR (driver(1)=grant)) THEN
          RETURN grant;
        ELSE RETURN request;
      END IF;
      ELSE RETURN free;
    END IF;
    ELSE RETURN free;
    END IF;
    RETURN free;
  END;
END;
ENTITY hsm_sender IS
  GENERIC (n: POSITIVE);
  PORT (me_ch_arbiter : INOUT pre_me_status_type;
        me_ch_sender : INOUT pre_me_status_type;
        recv_ev   : INOUT pre_event_type_arr (0 TO n-1));
END;

ARCHITECTURE hsm_sender_beh OF hsm_sender IS
BEGIN
  PROCESS
    VARIABLE ready: BOOLEAN;
  BEGIN
    IF me_ch_sender/= request THEN -- wait for request
      WAIT UNTIL me_ch_sender=request;
    END IF;
    me_ch_arbiter <= request; -- pass on request
    WAIT UNTIL me_ch_arbiter = grant; -- wait for grant
    me_ch_sender <= grant; -- pass on grant
    WAIT UNTIL me_ch_sender=free; -- wait for data present
    me_ch_sender <= free; -- withdraw grant
    FOR i IN recv_ev'RANGE LOOP -- data sent to all receivers
      recv_ev(i) <= (NOT recv_ev(i).private, TRUE);
    END LOOP;
    ready:= TRUE;
    FOR i IN recv_ev'RANGE LOOP -- test all receivers
      ready:= ready AND recv_ev(i).value;
    END LOOP;
    EXIT WHEN ready; -- only when all receivers acknowledged
    WAIT ON recv_ev;
    me_ch_arbiter <= free; -- release the channel
    WAIT UNTIL me_ch_arbiter= free;
  END PROCESS;
END;

USE WORK.if_pre_types.ALL;

PACKAGE tl_types IS
  -- **** general ****
  CONSTANT delay: Time := 1 NS; -- delay is used after each TL statement
  -- **** EVENTS ****
  SUBTYPE event_type IS pre_event_type;
  SUBTYPE event_type_arr IS pre_event_type_arr;
  -- **** MUTUAL EXCLUSION ****
  TYPE id_type IS (disconnected, connected, stors);
  SUBTYPE me_status_type IS pre_me_status_type;
  SUBTYPE me_status_type_arr IS pre_me_status_type_arr;

  PROCEDURE arbit (SIGNAL channels: INOUT me_status_type_arr);
  -- **** COMMUNICATION ****
  TYPE comm_status IS (free, data_present, acknowledge, no_data);
  TYPE comm_status_array IS ARRAY (NATURAL RANGE<>) OF comm_status;
  FUNCTION resolve_comm_status (driver: IN comm_status_array) RETURN comm_status;
  SUBTYPE comm_status_type IS resolve_comm_status comm_status;

  PROCEDURE wait_excl_channel (SIGNAL channel: INOUT me_status_type);
  PROCEDURE release_channel (SIGNAL channel: INOUT me_status_type);
COMPONENT hsm_sender
GENERIC (n: POSITIVE);
PORT (me_ch_arbiter : INOUT pre_me_status_type;
    me_ch_sender : INOUT pre_me_status_type;
    recv_ev : INOUT pre_event_type_arr (0 TO n-1));
END COMPONENT;
END ii_types;

PACKAGE BODY ii_types IS
  -- **** MUTUAL EXCLUSION ****
  PROCEDURE arbiter (SIGNAL channels: INOUT me_status_type_arr) IS
    VARIABLE previous: NATURAL;    -- number of last active process
    BEGIN
        previous := channels'HIGH;    -- initially test all processes
        LOOP
            FOR i IN channels'RANGE LOOP
                IF channels(i) = request THEN    -- request found
                    channels(i) <= grant;
                    WAIT UNTIL channels(i) = free;
                    channels(i) <= free;
                    previous := i;    -- make a full round
                    NEXT;
                END IF;
                IF i = previous THEN    -- made a full round, no request found
                    WAIT ON channels;
                END IF;
            END LOOP;
        END LOOP;
        -- **** COMMUNICATION ****
        FUNCTION resolve_comm_status (driver: IN comm_status_array) RETURN comm_status IS
            BEGIN
                -- signal can only have one or two drivers
                IF driver'LENGTH = 1 THEN RETURN driver(driver'LEFT);
                ELSIF driver'LENGTH = 2 THEN
                    IF ((driver(0)=data_present) OR (driver(1)=data_present)) THEN
                        IF ((driver(0)=free) OR (driver(1)=free)) THEN
                            THEN RETURN data_present;
                        ELSE RETURN acknowledge;
                    END IF;
                ELSE -- driver 0 or 1 must be 'no_data', or both are free (initial situation)
                    IF ((driver(0)=acknowledge) OR (driver(1)=acknowledge)) THEN
                        THEN RETURN no_data;
                    ELSE RETURN free;
                END IF;
                END IF;
                END IF;
                RETURN free;
            END;
        PROCEDURE wait_excl_channel (SIGNAL channel: INOUT me_status_type) IS
            BEGIN
                channel <= request;
                WAIT UNTIL channel = grant;
            END;
        PROCEDURE release_channel (SIGNAL channel: INOUT me_status_type) IS
            BEGIN
                channel <= free;
                WAIT UNTIL channel = free;
            END;
        END ii_types;
Appendix B: An example

This appendix demonstrates some of the capabilities of the current TL to VHDL compiler with an example. This appendix contains:

- Plan of the TL design ........................................ 88
- TL design file ............................................. 89
- Plan of the VHDL implementation .......................... 93
- VHDL design file ........................................... 94
- VHDL simulation results ................................... 102

In the plans a circle indicates a process and a rectangle is used to draw an entity. If an item has two names then the upper one is the identifier of the related declaration and the lower is the label that is attached to the usage of the item. Note that in entity 'first' the communication channel between process 'initiate' and process 'first_pipe' is an implicit channel named after the sending task. The lower process in the entity 'second' has no label, this is indicated with '<No_Name>'.

In the TL to VHDL compilation, two arbiters, two stores and a number of control signals have been added. These are made visible in the plan of the VHDL implementation.

The VHDL design file had to be manually adjusted because using the 'num_arr' alias in the procedure body generates an error when it is compiled in the VHDL simulator (see footnote in section 4.4). This adjustment is indicated by the row of stars.
Plan of the TL design
PACKAGE types IS
  TYPE nat_arr IS ARRAY (0 to 6) OF NATURAL;

  TYPE index_rec IS
    RECORD
      start: NATURAL;
      length: NATURAL;
    END RECORD;

  PROTOCOL pass_on_prot: RENDEZVOUS;
  PROTOCOL output_prot: HANDSHAKE;
END;

USE WORK.types.ALL;

ENTITY first IS
  PORT (ME_DATA num_arr: nat_arr;
        CHANNEL pass_on (first_send -> mid_recv): pass_on_prot;
        ME_DATA indexes: index_rec);
END;

ARCHITECTURE beh_first OF first IS
BEGIN
  initiate: PROCESS
    VARIABLE index: NATURAL;
    VARIABLE length: INTEGER;
    COM_DATA msg: NATURAL;
    BEGIN
      -- get message batch index information
      WAIT_EXCLUSION indexes;
      index:= indexes.start;
      length:= indexes.length;
      RELEASE indexes;
      LOOP
        -- get the message data
        WAIT_EXCLUSION num_arr;
        msg<= num_arr (index);
        RELEASE num_arr;
        -- put the data in the pipeline
        init_send: TASK BEGIN
        SEND msg TO first_recv;
        END TASK;
        -- prepare for next message
        index:= (index + 1) MOD 7;
        length:= length -1;
        -- finished this batch
        EXIT WHEN length<0;
      END LOOP;
    END PROCESS;

  first_pipe: PROCESS
    COM_DATA msg, data: NATURAL;
    BEGIN
      -- get message data to use
      first_recv: TASK BEGIN
        RECEIVE msg FROM init_send;
      END TASK;
      -- get the data
      WAIT_EXCLUSION num_arr;
      data<= num_arr (msg MOD 7);
      num_arr (msg MOD 7)<= (num_arr ((msg*3) MOD 7) + data) MOD 19;
      RELEASE num_arr;
    END PROCESS;
END;
-- send the data to the next pipeline element
first_send: TASK BEGIN
    SEND data TO mid_recv ON pass_on;
END TASK;
END PROCESS;
END;

USE WORK.types.ALL;

ENTITY second IS
    PORT (ME_DATA num_arr: nat_arr;
          CHANNEL pass_on (first_send -> mid_recv): pass_on_prot;
          CHANNEL output (mid_send -> last_recv): output_prot;
          ME_DATA indexes: index_rec);
END;

ARCHITECTURE beh_second OF second IS

PROCEDURE pipe_element (CHANNEL chan_in (previous_send -> this_recv): pass_on_prot;
                         ME_DATA num_arr: nat_arr;
                         CHANNEL chan_out (this_send -> next_recv): output_prot;
                         EVENT seven_fold)
IS
    BEGIN
    -- get message data to use
    this_recv: TASK BEGIN
        RECEIVE msg FROM previous_send ON chan_in;
    END TASK this_recv;

    -- get the data
    WAIT_EXCLUSION num_arr;
    data <= num_arr (msg MOD 7);
     num_arr (((msg + 1) MOD 7) <= (data * 3) MOD 19);
    RELEASE num_arr;

    IF (data MOD 7) = 0 THEN
    -- filter out seven folds
        SIGNAL_EVENT seven_fold;
    ELSE
    -- send the data to the next pipeline element
    this_send: TASK BEGIN
        SEND data TO next_recv ON chan_out;
    END TASK;
    END IF;
END;

EVENT seven_fold;
BEGIN
    -- a concurrent procedure call with positional and named association
    mid_pipe: pipe_element (pass_on (first_send -> mid_recv), num_arr,
                             chan_out (this_send -> next_recv) => output (mid_send -> last_recv),
                             sevenfold => seven_fold);

PROCESS
BEGIN
    -- wait for a seven fold to be read
    WAIT_EVENT seven_fold;

    -- adjust the batch start address
    WAIT_EXCLUSION indexes;
    indexes.start <= (indexes.start + 3) MOD 7;
    RELEASE indexes;
END PROCESS;
END;
ENTITY oscillating IS END;

USE WORK.types.ALL;

ARCHITECTURE beh_oscillating OF oscillating IS

COMPONENT first
  PORT (ME_DATA num_arr: nat_arr;
        CHANNEL pass_on (first_send -> mid_recv): pass_on_prot;
        ME_DATA indexes: index_rec);
END COMPONENT;

COMPONENT second
  PORT (ME_DATA num_arr: nat_arr;
        CHANNEL pass_on (first_send -> mid_recv): pass_on_prot;
        CHANNEL output (mid_send -> last_recv): output_prot;
        ME_DATA indexes: index_rec);
END COMPONENT;

ME_DATA num_arr: nat_arr := (3, 7, 17, 13, 5, 11, 1);
ME_DATA indexes: index_rec := (3, 5);

CHANNEL pass_on: pass_on_prot;
CHANNEL output: output_prot;

EVENT passed_27;
BEGIN

first_pipe: first PORT MAP (num_arr,
  pass_on (first_send -> second_recv),
  indexes);

second_pipe: second PORT MAP (num_arr,
  pass_on (first_send -> second_recv),
  output (second_send -> last_recv),
  indexes);

last_pipe: PROCESS
  COM_DATA msg: NATURAL;
  VARIABLE accu: NATURAL := 0;
BEGIN
  -- get message data to use
  last_recv: TASK BEGIN
    RECEIVE msg FROM second_send ON output;
  END TASK this_recv;

  -- accumulate the data
  accu := accu + msg;

  IF accu > 27 THEN
    -- indicate passed the 27 limit
    SIGNAL_EVENT passed_27;
    accu := 0;
  END IF;
END PROCESS last_pipe;

count_27: PROCESS
  VARIABLE count: NATURAL := 0;
BEGIN
  -- wait until the 27 limit is passed
  WAIT_EVENT passed_27;

  -- count these events
  count := count + 1;

  IF count = 3 THEN
    WAIT_EXCLUSION indexes;
    -- set a new batch length
    indexes.length := (indexes.length + 5) MOD 7;
IF indexes.length= 0 THEN
  -- minimum length
  indexes.length<=1;
  END IF;
  RELEASE indexes;
  END IF;
  END PROCESS;
END beh_oscillating;
Plan of the VHDL implementation
LIBRARY TLVHDL;
USE TLVHDL.t_types.ALL;
-- This next use clause is a work around for a VHDL simulator problem, the problem is
-- that the SUBTYPE 'me_status_type' defined in 'l_types' cannot be used here.
-- Instead 'pre_me_status_type' defined in 'l_pre_types' is used.
USE TLVHDL.t_pre_types.ALL;

PACKAGE special_types IS
  SUBTYPE NATURAL_ch_type IS NATURAL;

  TYPE nat_arr_2 IS
    ARRAY (0 TO 6) OF NATURAL;

  TYPE nat_arr_2_record IS
    RECORD
      data: nat_arr_2;
      id: id_type;
    END RECORD;

  TYPE nat_arr_2_record_array IS ARRAY(NATURAL RANGE<>) OF nat_arr_2_record;
  FUNCTION nat_arr_2_resolve (driver: IN nat_arr_2_record_array) RETURN nat_arr_2_record;
  SUBTYPE nat_arr_2_bus_type IS nat_arr_2_resolve nat_arr_2_record;

  PROCEDURE nat_arr_2_wait_excl (SIGNAL req-chan: INOUT pre_me_status_type;
                                SIGNAL data_bus: INOUT nat_arr_2_bus_type);

  PROCEDURE nat_arr_2_release (SIGNAL req-chan: INOUT pre_me_status_type;
                                SIGNAL data_bus: INOUT nat_arr_2_bus_type);

  TYPE index_rec_1 IS
    RECORD
      start: NATURAL;
      length: NATURAL;
    END RECORD;

  TYPE index_rec_1_record IS
    RECORD
      data: index_rec_1;
      id: id_type;
    END RECORD;

  TYPE index_rec_1_record_array IS ARRAY(NATURAL RANGE<>) OF index_rec_1_record;
  FUNCTION index_rec_1_resolve (driver: IN index_rec_1_record_array) RETURN index_rec_1_record;
  SUBTYPE index_rec_1_bus_type IS index_rec_1_resolve index_rec_1_record;

  PROCEDURE index_rec_1_wait_excl (SIGNAL req-chan: INOUT pre_me_status_type;
                                   SIGNAL data_bus: INOUT index_rec_1_bus_type);

  PROCEDURE index_rec_1_release (SIGNAL req-chan: INOUT pre_me_status_type;
                                  SIGNAL data_bus: INOUT index_rec_1_bus_type);
END;

PACKAGE BODY special_types IS

  FUNCTION nat_arr_2_resolve (driver: IN nat_arr_2_record_array)
    RETURN nat_arr_2_record IS
    BEGIN
      temp:=driver(driver'LEFT);
      FOR i IN driver'RANGE LOOP
        IF driver(i).id = connected THEN RETURN driver(i); END IF;
        IF driver(i).id = store THEN temp:=driver(i); END IF;
      END LOOP;
      RETURN temp;
    END;

  PROCEDURE nat_arr_2_wait_excl (SIGNAL req-chan: INOUT pre_me_status_type;
                                 SIGNAL data_bus: INOUT nat_arr_2_bus_type) IS
    BEGIN
      req-chan <= request;
      WAIT UNTIL req-chan = grant;
data_bus <= (data_bus.data, connected);
WAIT FOR delay;
END;

PROCEDURE nat_err_2_release (SIGNAL req_chan: INOUT pre_me_status_type;
SIGNAL data_bus: INOUT nat_err_2_bus_type) IS
BEGIN
WAIT FOR 0 NS;
data_bus.id <= disconnected;
req_chan <= free;
WAIT FOR delay;
END;

FUNCTION index_rec_1_resolve (driver: IN index_rec_1_record_array)
RETURN index_rec_1_record IS
VAR temp: temp: index_rec_1_record;
BEGIN
VAR temp: temp: temp:=drivw(driw(LEFT);
FOR i IN driver(RANGE LOOP
IF driver(i).id = connected THEN RETURN driver(i); END IF;
IF driver(i).id = store THEN temp:=driver(i); END IF;
END LOOP;
RETURN temp;
END;

PROCEDURE index_rec_1_wait_excl (SIGNAL req_chan: INOUT pre_me_status_type;
SIGNAL data_bus: INOUT index_rec_1_bus_type) IS
BEGIN
req_chan <= request;
WAIT UNTIL req_chan = grant;
data_bus <= (data_bus.data, connected);
WAIT FOR delay;
END;

PROCEDURE index_rec_1_release (SIGNAL req_chan: INOUT pre_me_status_type;
SIGNAL data_bus: INOUT index_rec_1_bus_type) IS
BEGIN
WAIT FOR 0 NS;
data_bus.id <= disconnected;
req_chan <= free;
WAIT FOR delay;
END;

LIBRARY TLVHDL;
USE TLVHDL_all;
USE WORK_special_types.ALL;

PACKAGE types IS
-- TYPE nat_arr IS ARRAY 0 TO 6 OF NATURAL;
SUBTYPE nat_arr IS nat_arr;

-- TYPE index_rec IS
-- RECORD
-- start: NATURAL;
-- length: NATURAL;
-- END RECORD;
SUBTYPE index_rec IS index_rec;

-- PROTOCOL pass_on_prot: RENDEZVOUS;
-- PROTOCOL output_prot: HANDSHAKE;
END;

USE WORK_types.ALL;

LIBRARY TLVHDL;
ENTITY first IS
PORT (-- ME_DATA num: nat_arr;  
    SIGNAL num_me: OUT me_status_type_arr (0 TO 1);  
    SIGNAL num_bus: OUT nat_arr_2_bus_type;  
    -- CHANNEL pass_on (first_send -> mid_recv): pass_on_prot;  
    SIGNAL pass_on: OUT NATURAL_ch_type;  
    SIGNAL first_send_mid_recv: OUT comm_status_type;  
    -- ME_DATA indexes: indexes_recv;  
    SIGNAL indexes_recv: OUT me_status_type_arr (0 TO 0);  
    SIGNAL indexes_bus: OUT indexes_recv_1_bus_type);  
ALIAS num_arr: nat_arr IS num_arr_bus.data;  
ALIAS indexes: indexes_recv IS indexes_recv_bus.data;  
END;
ARCHITECTURE beh_first OF first IS  
SIGNAL init_send: NATURAL_ch_type;  
SIGNAL init_send_first_recv: event_type;  
BEGIN  
itinitiate: PROCESS  
    VARIABLE index: NATURAL;  
    VARIABLE length: INTEGER;  
    -- COM_DATA msg: NATURAL;  
    VARIABLE msg: NATURAL;  
BEGIN  
    -- get message batch index information  
    -- WAIT_EXCLUSION indexes;  
    index_rec_1_wait_excl (indexes_recv_me_ch(0), indexes_recv_bus);  
    index:= indexes.start;  
    length:= indexes.length;  
    -- RELEASE indexes;  
    index_rec_1_release (indexes_recv_me_ch(0), indexes_recv_bus);  
LOOP  
    -- get the message data  
    -- WAIT_EXCLUSION num_arr;  
    natarr_2_wait_excl (num_arr_recv_me_ch(0), num_arr_bus);  
    -- msg< num_arr (index);  
    msg:= num_arr (index);  
    -- RELEASE num_arr;  
    natarr_2_release (num_arr_recv_me_ch(0), num_arr_bus);  
    -- put the data in the pipeline  
    -- init_send: TASK BEGIN  
    -- SEND msg TO first_recv;  
    IF init_send_first_recv.value THEN  
        WAIT UNTIL NOT (init_send_first_recv.value);  
    END IF;  
    init_send <= msg;  
    init_send_first_recv <= (NOT init_send_first_recv.private, TRUE);  
    WAIT FOR delay;  
    -- END TASK;  
    -- prepare for next message  
    index:= (index + 1) MOD 7;  
    length:= length -1;  
    -- finished this batch  
    EXIT WHEN length<0;  
END LOOP;  
END PROCESS;
first_pipe: PROCESS
-- COM DATA msg, data: NATURAL;
VARIABLE msg, data: NATURAL;

BEGIN
-- get message data to use
-- first_recv: TASK BEGIN
  -- RECEIVE msg FROM init_send;
  IF NOT init_send_first_recv.value THEN
    WAIT UNTIL init_send_first_recv.value;
  END IF;
  msg := init_send;
  init_send_first_recv <= (init_send_first_recv.private, FALSE);
  WAIT FOR delay;
-- END TASK;

-- get the data
-- WAIT EXCLUSION num_arr;
  nat_arr_2_wait_excl (num_arr_me_ch(1), num_arr_bus);
  data <= num_arr (msg MOD 7);
  data := num_arr (msg MOD 7);
  num_arr (msg MOD 7) <= (num_arr ((msg+3) MOD 7) + data) MOD 19;
  -- RELEASE num_arr;
  nat_arr_2_release (num_arr_me_ch(1), num_arr_bus);

-- send the data to the next pipeline element
-- first_send: TASK BEGIN
  -- SEND data TO mic_recv ON pass_on;
  IF first_send_mid_recv=free THEN
    WAIT UNTIL (first_send_mid_recv=free);
  END IF;
  pass_on <= data;
  first_send_mid_recv <= data_sent;
  WAIT FOR 0 NS;
  LOOP
    IF first_send_mid_recv=acknowledge THEN
      first_send_mid_recv <= no_data;
    END IF;
    EXIT WHEN (first_send_mid_recv=data_sent);
    WAIT ON first_send_mid_recv;
  END LOOP;
  WAIT FOR delay;
-- END TASK;

END PROCESS;
END;

USE WORK.types.ALL;
LIBRARY TLVHDL;
USE TLVHDL.types.ALL;
USE WORK.special_types.ALL;

ENTITY second IS
  PORT (-- ME_DATA num_arr: nat_arr;
    SIGNAL num_arr_me_ch: INOUT me_status_type_arr (0 TO 0);
    SIGNAL num_arr_bus: INOUT nat_arr_2_bus_type;
    -- CHANNEL pass_on (first_send -> mid_recv): pass_on_prot;
    SIGNAL pass_on: IN NATURAL_ch_type;
    SIGNAL first_send_mid_recv: INOUT comm_status_type;
    -- CHANNEL output (mid_send -> last_recv): output_prot;
    SIGNAL output: OUT NATURAL_ch_type;
    SIGNAL mid_send_last_recv: INOUT event_type;
    -- ME_DATA indexes: index_rec;
    SIGNAL indexes_me_ch: INOUT me_status_type_arr (0 TO 0);
    SIGNAL indexes_bus: INOUT index_rec_1_bus_type);

ALIAS num_arr: nat_arr IS num_arr_bus.data;
ALIAS indexes : indexes_rec IS indexes_bus.data;
END;

ARCHITECTURE beh_second OF second IS

PROCEDURE pipe_element (-- CHANNEL chan_in (previous_send -> this_recv): pass_on_prot;
  SIGNAL chan_in: IN NATURAL_ch_type;
  SIGNAL previous_send_this_recv: INOUT comm_status_type;
  -- ME_DATA num_arr: nat_arr;
  SIGNAL num_arr_me_ch: INOUT nat_arr_status_type_arr (0 TO 0);
  SIGNAL num_arr_bus: INOUT nat_arr_2_bus_type;
  -- CHANNEL chan_out (this_send -> next_recv): output_prot;
  SIGNAL chan_out: OUT NATURAL_ch_type;
  SIGNAL this_send_next_recv: INOUT event_type;
  -- EVENT seven_fold;
  SIGNAL seven_fold: (INOUT event_type) IS
    -- COM_DATA msg, data: NATURAL;
    VARIABLE msg, data: NATURAL;
    VARIABLE seven_fold_var: BOOLEAN;
    ALIAS num_arr: nat_arr IS num_arr_bus.data;
BEGIN
  -- get message data to use
  -- this_recv: TASK BEGIN
  -- RECEIVE msg FROM previous_send ON chan_in;
  IF previous_send_this_recv=data_present THEN
    WAIT UNTIL previous_send_this_recv=data_present;
  END IF;
  msg := chan_in;
  previous_send_this_recv <= acknowledge;
  WAIT UNTIL previous_send_this_recv=no_data;
  previous_send_this_recv <= free;
  WAIT FOR delay;
  -- END TASK this_recv;

  -- get the data
  -- WAIT EXCLUSION num_arr;
  nat_arr_2_wait_excl (num_arr_me_ch(0), num_arr_bus);
  -- data<= num_arr (msg MOD 7);
  data:= num_arr (msg MOD 7);
  -- num_arr ((((msg +1) MOD 7)* (data * 3) MOD 19;

  num_arr_bus.data ((((msg +1) MOD 7)* (data * 3) MOD 19;
  -- RELEASE num_arr;
  nat_arr_2_release (num_arr_me_ch(0), num_arr_bus);

  IF (data MOD 7)= 0 THEN
    -- filter out seven folds
    -- SIGNAL_EVENT seven_fold;
    seven_fold_var:= FALSE;
    LOOP
      IF NOT seven_fold_var AND NOT seven_fold.value THEN
        seven_fold <= (NOT seven_fold.private, TRUE);
        seven_fold_var:= TRUE;
      END IF;
      EXIT WHEN seven_fold_var;
      WAIT ON seven_fold;
    END LOOP;
    WAIT FOR delay;
  ELSE
    -- send the data to the next pipeline element
    -- this_send: TASK BEGIN
    -- SEND data TO next_recv ON chan_out;
    IF this_send_next_recv.value THEN
      WAIT UNTIL NOT (this_send_next_recv.value);
    END IF;
    chan_out <= data;
    this_send_next_recv <= (NOT this_send_next_recv.private, TRUE);
  END IF;
END beh_second;
BEGIN
-- a concurrent procedure call with positional and named association
mid_pipe: pipe_element (- pass_on (first_send -> mid_recv)
  pass_on, first_send_mid_recv, -- num_arr
  num_arr_me_ch(0 TO 0), num_arr_bus,
  -- chan_out (this_send -> next_recv) => output (mid_send -> last_recv)
  chan_out => output, this_send_next_recv => mid_send_last_recv,
  seven_fold => seven_fold);

PROCESS
BEGIN
-- wait for a seven fold to be read
-- WAIT_EVENT seven_fold;
IF NOT (seven_fold.value) THEN
  WAIT UNTIL seven_fold.value;
END IF;
IF seven_fold.value THEN
  seven_fold <= (seven_fold.private, FALSE); END IF;
  WAIT FOR delay;

-- adjust the batch start address
-- WAIT_EXCLUSION indexes;
index_rec_1_wait_excl (indexes_me_ch(O), indexes_bus);
  indexes.start<= (indexes.start + 3) MOD 7;
-- RELEASE indexes;
index_rec_1_release (indexes_me_ch(O), indexes_bus);
END PROCESS;
END;

LIBRARY TLVHDL;
USE TLVHDL.itypes.ALL;
USE WORKspec:ialtypes.ALL;

ENTITY oscilating IS
  END;

USE WORK.types.ALL;

ARCHITECTURE beh_oscilating OF oscilating IS
  COMPONENT first
    PORT (-- ME_DATA num_arr: nat_arr;
      SIGNAL num_arr_me_ch: OUT me_status_type_arr (0 TO 1);
      SIGNAL num_arr_bus: INOUT nat_arr_2_bus_type;
    - CHANNEL pass_on (first_send -> mid_recv): pass_on_prot;
      SIGNAL pass_on: OUT NATURAL_ch_type;
      SIGNAL first_send_mid_recv: INOUT comm_status_type;
    - ME_DATA indexes: index_rec;
      SIGNAL indexes_me_ch: INOUT me_status_type_arr (0 TO 0);
      SIGNAL indexes_bus: INOUT index_rec_1_bus_type);
  END COMPONENT;

  COMPONENT second
    PORT (-- ME_DATA num_arr: nat_arr;
      SIGNAL num_arr_me_ch: INOUT me_status_type_arr (0 TO 0);
      SIGNAL num_arr_bus: INOUT nat_arr_2_bus_type;
    - CHANNEL pass_on (first_send -> mid_recv): pass_on_prot;
      SIGNAL pass_on: IN NATURAL_ch_type;
SIGNAL first_send_mid_recv: INOUT comm_status_type;
-- CHANNEL output (mid_send -> last_recv); output_prot;
SIGNAL output: OUT NATURAL_ch_type;
-- SIGNAL mid_send_last_recv: INOUT event_type;
-- ME_DATA indexes: index_recv;
SIGNAL indexes_me_ch: INOUT me_status_type_arr (0 TO 0);
SIGNAL indexes_bus: INOUT index_rec_1_bus_type);
END COMPONENT;
-- ME_DATA num_arr: nat_arr := (3, 7, 17, 13, 5, 11, 1);
SIGNAL num_arr_me_ch: me_status_type_arr (0 TO 2);
SIGNAL num_arr_bus: nat_arr_2_bus_type := (3, 7, 17, 13, 5, 11, 1), disconnected);
ALIAS num_arr: nat_arr IS num_arr_bus.data;
-- ME_DATA indexes: index_recv := (3, 3);
SIGNAL indexes_me_ch: me_status_type_arr (0 TO 2);
SIGNAL indexes_bus: index_rec_1_bus_type := (3, 5), disconnected);
ALIAS indexes: index_recv IS indexes_bus.data;
-- CHANNEL pass_on: pass_on_prot;
SIGNAL pass_on: NATURAL_ch_type;
SIGNAL first_send_second_recv: comm_status_type;
-- CHANNEL output: output_prot;
SIGNAL output: NATURAL_ch_type;
SIGNAL second_send_last_recv: event_type;
-- EVENT passed_27;
SIGNAL passed_27: event_type;
BEGIN
first_pipe: first PORT MAP (num_arr
num_arr_me_ch(0 TO 1), num_arr_bus,
-- pass_on (first_send -> second_recv)
pass_on, first_send_second_recv,
-- indexes
indexes_me_ch(0 TO 0), indexes_bus);
second_pipe: second PORT MAP (num_arr
num_arr_me_ch(2 TO 2), num_arr_bus,
-- pass_on (first_send -> second_recv)
pass_on, first_send_second_recv,
-- output (second_send -> last_recv)
output, second_send_last_recv,
-- indexes
indexes_me_ch(1 TO 1), indexes_bus);
last_pipe: PROCESS
-- COM_DATA msg: NATURAL;
VARIABLE msg: NATURAL;
VARIABLE accu: NATURAL := 0;
VARIABLE passed_27_var: BOOLEAN;
BEGIN
-- get message data to use
-- last_recv: TASK BEGIN
-- RECEIVE msg FROM second_send ON output;
IF NOT second_send_last_recv.value THEN
  WAIT UNTIL second_send_last_recv.value;
END IF;
msg := output;
second_send_last_recv <= (second_send_last_recv.private, FALSE);
WAIT FOR delay;
-- END TASK this_recv;

-- accumulate the data
accu := accu + msg;
IF accu = 27 THEN
VHDL design file

-- indicate passed the 27 limit
-- SIGNAL_EVENT passed_27;
passed_27_var := FALSE;
LOOP
  IF NOT passed_27_var AND NOT passed_27.value THEN
    passed_27 <= (NOT passed_27.private, TRUE);
passed_27_var := TRUE;
  END IF;
  EXIT WHEN passed_27_var;
  WAIT ON passed_27;
END LOOP;
  WAIT FOR delay;
accu := 0;
END IF;
END PROCESS last_pipe;

count_27: PROCESS
  VARIABLE count:NATURAL :=0;
BEGIN
  -- wait until the 27 limit is passed
  -- WAIT_EVENT passed_27;
  IF NOT (passed_27.value) THEN
    WAIT UNTIL passed_27.value;
  END IF;
  IF passed_27.value THEN passed_27 <= (passed_27.private, FALSE); END IF;
  WAIT FOR delay;

  -- count these events
  count:= count+1;
  IF count= 3 THEN
    -- WAIT_EXCLUSION indexes;
    index_rec_1_wait_excl (indexes_me_ch(2), indexes_bus);
    -- set a new batch length
    indexes.length<= (indexes.length+ 5) MOD 7;
    IF indexes.length= 0 THEN
      -- minimum length
      indexes.length<=1;
    END IF;
    -- RELEASE indexes;
    index_rec_1_release (indexes_me_ch(2), indexes_bus);
  END IF;
END PROCESS;

num_arr_arbit: arbit(num_arr_me_ch);
num_arr_store: num_arr_bus <= (num_arr, store);

indexes_arbit: arbit(indexes_me_ch);
indexes_store: indexes_bus <= (indexes, store);
END beh_oscilating;
In order to fit these results on the page all the identifiers of enumeration type have been abbreviated to two characters these are:

<table>
<thead>
<tr>
<th>tr</th>
<th>co</th>
<th>fr</th>
<th>dp</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>connected</td>
<td>free</td>
<td>data_present</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fa</th>
<th>di</th>
<th>re</th>
<th>nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>disconnected</td>
<td>request</td>
<td>no_data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>st</th>
<th>gr</th>
<th>ac</th>
</tr>
</thead>
<tbody>
<tr>
<td>store</td>
<td>grant</td>
<td>acknowledge</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>first_send</th>
<th>second_recv</th>
<th>init_send</th>
<th>first_recv</th>
</tr>
</thead>
<tbody>
<tr>
<td>pass_on</td>
<td>output</td>
<td>passed</td>
<td>sevenfold</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>num_arr_me_ch</th>
<th>num_arr_bus</th>
<th>indexes_me_ch</th>
<th>indexes_bus</th>
<th>init_send_first_recv</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns data</td>
<td>[second_send_last_recv]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>(3, 7, 13, 5, 11, 1, d)</td>
<td>(fr, fr, fr)</td>
<td>(tr, fr, fr)</td>
<td>(3, 5)</td>
</tr>
<tr>
<td>num_arr_me_ch</td>
<td>num_arr_bus</td>
<td>indexes_me_ch</td>
<td>pass_on</td>
<td>output</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>---------------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>16  +2</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>17  +3</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>18  +4</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>19  +1</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>20  +2</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>21  +3</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>22  +4</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>23  +1</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>24  +2</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>25  +3</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>26  +4</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>27  +1</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>28  +2</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>29  +3</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>30  +4</td>
<td>(fr, fr, fr)</td>
<td>(7, 2, 13, 1, 15, 5, 6)</td>
<td>st</td>
<td>st</td>
</tr>
</tbody>
</table>

VHDL simulation results
<table>
<thead>
<tr>
<th>num_err_me_ch</th>
<th>num_err_bus</th>
<th>indexes_me_ch</th>
<th>pass_on</th>
<th>output</th>
<th>passed?</th>
<th>seven_fold</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>35</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>36</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>37</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>38</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>40</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>41</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>42</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>43</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>44</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>45</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>46</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>48</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>49</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
<tr>
<td>50</td>
<td>(fr, fr, fr)</td>
<td>((10, 5, 2, 15, 1, 15), 7)</td>
<td>st</td>
<td>(fr, fr, fr)</td>
<td>(6, 5)</td>
<td>st</td>
</tr>
</tbody>
</table>

**VHDL simulation results**
INDEX

:= 10
<= 10
=> 10
Abstract syntax 21
ALL 12
Arbiter 35
Architecture body 9
Association
        named 10
        positional 10
Ast 22
Attributes 20
Behaviour
        of entity 9
Buffer for communication 72
Bus emulation 34
CDA 21
Cg 22
Channel 42
Code generator 21
Communication 40
Compiler 19
Component declaration 10
Components 9
Concrete syntax 20
Context dependent analyzer 21
Data bus 15
Deadlock 32
Declaration part 14
Declarative region 14
Default protocol 42, 65
Delay, constant 25
Design unit 12
Directly visible 14
Disconnect driver 16
Driver 15
Embedded system 17
Entity 9
Environment tree 21
Event object 26
Explicit subchannels 42
Fairness 32
FIFO 75
Functional descriptions 9

Generic clause 49
GMD-toolbox 21
Hidden 14
Hierarchical arbiter 68
Highest hierarchical level 11
HSM-sender process 47
HSM-senders list 64
HSM-status-signal 47
IdPrefix 56
Implicit channel 42
Implicit protocol 65
Implicit subchannels 42
In-line code 30
Indirectly visible 15
INOUT 17
Intermediate types 53, 66
Lalr 22
Leaf-task 18
Local
        communication 11
        declaration 10
        process 11
Lockstep 43
Mark 58
Merged channel 42
Merged protocol 42
Message object 41
Mode 10
Multidrop channel 42
Mutual exclusion 32
Named association 10
Nested 14
NULL 16
Packages 12
Parser 20
Parser tree 20
Peak load process 72
Positional process 72
Positional process 72
Priority mutual exclusion 70
Procedural language 9
Process 9
Puma 21, 22
qq_. 56
Resolution function 15
Rex 22
Rpp 22
Scanner 19
Scope 14
Secondary unit 12
Semantics 21
Signal 9
Signal declaration 10
Special_types 34, 52
Starvation 32
Statement part 14
Status-signal 35
Store process 34
Strong typing 12
Structural description 9
Subchannel 42
Subprograms 12
Subtype declaration 16
Subtype indication 58
Subtypes 12
Target types 53, 66
Time 18
Time unit 18
TL 7
TL specific statement 18
TL_TYPES 25
TLVHDL 7
Token 19, 69
Token-ring mutual exclusion 69
Type list 55, 60, 63
Use clause 12
VHDL 9
Visible

directly 14
Word 19
WORK 12