Component-Based Software Engineering model using Simulink

A dissertation submitted to The University of Manchester for the degree of MSc in the Faculty of Engineering and Physical Science

2009

Patrice T. SIATCHOUA

School of Computer Science
Contents

LIST OF TABLES .......................................................................................................... 4
LIST OF FIGURES ........................................................................................................ 5
LIST OF ABBREVIATIONS ........................................................................................ 6
ABSTRACT ..................................................................................................................... 7
DECLARATION ............................................................................................................. 8
COPYRIGHT STATEMENT ........................................................................................ 9
ACKNOWLEDGMENTS ............................................................................................ 10
1 INTRODUCTION ................................................................................................. 11
2 BACKGROUND ................................................................................................... 14
  2.1 THE COMPONENT PARADIGM ............................................................................. 14
  2.2 COMPOSITION OPERATORS: EXOGENOUS CONNECTORS .................................. 16
  2.3 ELEMENTS OF MATLAB AND SIMULINK .......................................................... 18
    2.3.1 MATLAB .................................................................................................... 18
    2.3.2 Simulink ..................................................................................................... 18
3 DESIGN CONSIDERATIONS ............................................................................ 20
  3.1 DESIGN USING EMBEDDED MATLAB ............................................................ 20
    3.1.1 What is embedded MATLAB ..................................................................... 20
    3.1.2 Advantages of design based on embedded MATLAB ................................ 21
    3.1.3 Constraints of design based on embedded MATLAB ............................... 21
  3.2 SOME LIMITATIONS OF SIMULINK............................................................... 23
  3.3 SIGNALS, BUSES AND STRUCTURES IN A SIMULINK ENVIRONMENT ............ 23
  3.4 THE CONCEPT OF LIBRARY IN SIMULINK ..................................................... 24
4 DESIGN AND IMPLEMENTATION ................................................................ 25
  4.1 GENERAL REQUIREMENTS ON THE DESIGN OF COMPONENT .................... 25
  4.2 DEFINITION OF INPUT STRUCTURE ............................................................... 26
  4.3 DEFINITION OF OUTPUT STRUCTURE ............................................................ 29
  4.4 ERROR HANDLING .......................................................................................... 31
  4.5 ATOMIC COMPONENT ....................................................................................... 32
  4.6 COMPOSITION OPERATORS ............................................................................ 35
    4.6.1 Pipe connector .......................................................................................... 35
    4.6.2 Sequencer connector ............................................................................... 41
    4.6.3 Selector connector .................................................................................... 43
  4.7 COMPONENT REPOSITORY .............................................................................. 45
5 TESTING ............................................................................................................... 47
  5.1 MATLAB TEST FILES .................................................................................... 48
  5.2 CONFIGURING AND RUNNING THE SIMULATION ........................................... 49
  5.3 ANALYSING THE RESULTS ............................................................................. 50
    5.3.1 Atomic component ..................................................................................... 50
    5.3.2 Composite component using a pipe connector ........................................ 51
    5.3.3 Composite component using a sequencer connector ............................... 52
5.3.4 Composite component using a selector connector ........................................ 53
5.3.5 Composite component using two pipe connectors ..................................... 53
5.3.6 Composite component using a pipe and a sequencer connectors ............. 54

6 CONCLUSION .................................................................................................................. 55

6.1 SUMMARY OF ACHIEVEMENTS AND CRITICAL ANALYSIS ........................... 55
6.2 FUTURE WORKS ........................................................................................................ 56

APPENDIX A: ATOMIC COMPONENT SOURCE CODES ........................................... 57

APPENDIX B: PIPE CONNECTOR SOURCE CODES .................................................. 61

LISTING B1 .................................................................................................................... 61
LISTING B2 .................................................................................................................... 62
LISTING B3 .................................................................................................................... 64
LISTING B4 .................................................................................................................... 65
LISTING B5 .................................................................................................................... 65

APPENDIX C: SEQUENCER CONNECTOR SOURCE CODES .................................... 67

LISTING C1 .................................................................................................................... 67
LISTING C2 .................................................................................................................... 69
LISTING C3 .................................................................................................................... 70
LISTING C4 .................................................................................................................... 70

APPENDIX D: SELECTOR CONNECTOR SOURCE CODES .................................... 73

LISTING D1 .................................................................................................................... 73
LISTING D2 .................................................................................................................... 75
LISTING D3 .................................................................................................................... 76

APPENDIX E: LOOP SOURCE CODES ....................................................................... 77

LISTING E1 .................................................................................................................... 77
LISTING E2 .................................................................................................................... 78
LISTING E3 .................................................................................................................... 78

APPENDIX F: INPUT AND OUTPUT STRUCTURE DEFINITION ............................ 80

APPENDIX G: MATLAB PROCESSING FILE .............................................................. 83

REFERENCES .................................................................................................................. 85
List of Tables

Table 1: Data Type-Uint8 correspondence ................................................................. 27
Table 2: Inputs structure field description ................................................................. 28
Table 3: Outputs structure field description ............................................................. 30
List of Figures

Figure 1: Idealised component life cycle (Lau, Ling, & Wang, 2006) .............................................................. 16
Figure 2: Exogenous connection .......................................................................................................................... 17
Figure 3: Simulink environment (Mathworks, 2009) .......................................................................................... 19
Figure 4: Schematic of a Component showing inputs and outputs ports ......................................................... 25
Figure 5: Component with the method name and inputs parameters ................................................................. 26
Figure 6: Internal representation of input parameters and method name .......................................................... 28
Figure 7: C3 = pipe (C1, C2) .................................................................................................................................. 29
Figure 8: Error path illustration ............................................................................................................................ 31
Figure 9: Function Call Subsystem used to implement atomic component ....................................................... 32
Figure 10: Atomic component for a bank account (part 1) ................................................................................. 33
Figure 11: Atomic component for a bank account (part 2) ............................................................................... 33
Figure 12: Atomic component for a bank account (part 3) ............................................................................... 34
Figure 13: Pipe connector using a Simulink subsystem block and a Stateflow chart ........................................ 36
Figure 14: Pipe connector with two atomic components .................................................................................... 37
Figure 15: Control state for connectors ............................................................................................................... 38
Figure 16: Description state .................................................................................................................................. 39
Figure 17: Pipe methods matching ....................................................................................................................... 39
Figure 18: Computation state for a pipe ................................................................................................................ 40
Figure 19: Computing state of the pipe connector ............................................................................................. 41
Figure 20: Sequencer connector top view showing the ports ............................................................................. 42
Figure 21: Sequencer method matching ............................................................................................................. 43
Figure 22: Selector connector top level view and configuration window ............................................................ 44
Figure 23: Selector method matching .................................................................................................................. 44
Figure 24: Computing sub state for a selector .................................................................................................... 45
Figure 25: Component repository ......................................................................................................................... 46
Figure 26: Test environment .................................................................................................................................. 47
Figure 27: Test configuration .................................................................................................................................. 48
Figure 28: MATLAB configuration file ................................................................................................................ 49
Figure 29: Simulink configuration parameters .................................................................................................... 49
Figure 30: Simulink configuration parameters- chart options ............................................................................. 50
Figure 31: Result of testing an atomic component ............................................................................................... 51
Figure 32: Result of testing a composite component using a pipe ...................................................................... 52
Figure 33: Result of testing a composite component using a sequencer ............................................................. 52
Figure 34: Result of testing a composite component using a selector .................................................................. 53
Figure 35: Result of testing a composite component using two pipes .................................................................. 54
Figure 36: Result of testing a composite component using a pipe and a sequencer ........................................ 54
List of Abbreviations

EML: Embedded MATLAB
Abstract

Component based software engineering aims to tackle the ever predominant issue of software reuse. This software development methodology is increasingly being used in many engineering domains, with the hope of effectively cutting the time to market created products. Such a component based development framework, developed at the University of Manchester in the software component group, is the component model based on an exogenous connector. It aims to tackle shortcomings of existing software component models by introducing connectors that effectively encapsulate control, data and computation.

In addition to using software methodology, such as the one based on the component paradigm to promote reuse and decrease time to market, engineers make use of effective tools that accelerate the design of a new system. One of these tools is ‘Simulink’, a tool that is widely used across many fields ranging from financial to aerospace applications.

This work presents an approach to implement the component model based on an exogenous connector in Simulink. Through using a combination of wisely chosen Simulink artefacts and appropriate Stateflow charts, it is shown that it is possible to implement the core connectors underpinning this model. We used an example based on a bank system to demonstrate the validity of our approach.
Declaration

No portion of the work referred to in this dissertation, has been submitted in support of an application for another degree or qualification of this or any other university, or other institute of learning.
Copyright Statement

i. Copyright in text of this dissertation rests with the author. Copies (by any process) either in full, or of extracts, may be made only in accordance with instructions given by the author. Details may be obtained from the appropriate Graduate Office. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the author.

ii. The ownership of any intellectual property rights which may be described in this dissertation is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

iii. Further information on the conditions under which disclosures and exploitation may take place is available from the Head of the School of Computer Science.
Acknowledgments

I would like to take this opportunity to first and foremost thank my supervisor Dr. Kung-Kiu Lau for giving me, through this topic, an insight into component based software engineering, and for allowing me to participate in numerous workshops, organised in the software component group. I have undoubtedly gained knowledge and insight through this experience. I would also like to thank all members of the group for their valuable discussions and debate and the many individuals who have directly or indirectly contributed to this dissertation work.
1 Introduction

The ever important status software receives within many developmental systems need no longer be demonstrated. This observation is apparent within the field of telecommunications systems and consumer electronics, as well as many other engineering applications. One immediate consequence is that this has, arguably, frequently made software the “weakest link” in these different systems. This is particularly evident, within the complexity or reliability of time to market (TTM) systems. There have been in the past numerous reports of system projects which have failed due to a problem linked in some way to software development. At the same time, this increasing role of software within system development can also be seen as an opportunity for increase competiveness as this can constitute in many systems the added value.

One software engineering approach, which is believed to be capable of tackling many of the issues associated with software development, is component based software engineering (CBSE). The principle behind CBSE is to build applications by associating components that provide basic functionalities. Those components are then linked together in conceptually the same way as electronics components are linked together on a printed circuit board (PCB).

The notion of a component is not new, its initial use can be dated back to 1968 (Mathworks, 2009). However its widespread usage really began with the industrial models of Enterprise JavaBeans (EJB) by Sun Microsystems and Component Object Model (COM) technologies by Microsoft in the 1990s.

Nevertheless, defining the notion of a component has not being a straightforward task. Szyperski (1998) could account in 1997 for 11 different definitions of a component and proposed one that seems to be widely accepted:
‘A component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties’

This definition, while informal, stresses a number of important aspects of a component. Firstly, a component is contractually specified by the services it provides and the services it requires. The provided component services can only be accessed through its interfaces and not by directly accessing the internal structure of the component. Secondly, a component is a deliverable unit and as such has characteristics of an executable package of software (Brown, 2000).

This mode of software development is believed to be highly suitable for the very important property of software reusability, compared to other software development approaches such as object oriented programming (OOP).

The aim of this dissertation is to implement a component based software engineering approach based on exogenous connectors in Simulink. Simulink is a commercial tool developed by The Mathworks for modelling, analysing and simulating dynamic systems. It is widely used across many industries but is especially employed in application domains such as automotive and aerospace systems.

Stated objectives are:

- Develop an approach to create atomic component in Simulink

- Develop the component based engineering approach based on the exogenous connectors; this will result in developing a number of core connectors used in this model, namely the pipe, the sequencer and the selector connectors. The development of these connectors will be built having reusability in mind.

- Develop a repository of component and connectors in Simulink that can be used to construct new components.
• Develop a testing environment that will be used to analyse the developed artefacts

The scope of this project does not however, include the development of a particular system although an example system will be used to demonstrate the behaviour of the developed artefacts.

The remainder of this report is organised as follows: a background section (Chapter 2) briefly introducing software components and the idea behind exogenous connections, as well as a brief introduction to MATLAB and Simulink. The next chapter (Chapter 3) investigates the factors that would influence any design and implementation decisions. Chapter 4 describes in more detail elements of the design and implementation. Chapter 5 shows the result of testing the developed artefacts followed by the conclusion in Chapter 6.
2 Background

2.1 The component paradigm

Reusability has always been the ultimate aim of any software engineering approach. Procedural programming paradigm as well as object oriented programming paradigm has each used a different approach to get closer to that ultimate aim. These approaches have however been inconsistent in their success. One approach that has been around for a while, with the main aim of always been reusability, is based on the notion of component and subsequently on the notion of composition.

At the beginning of this paradigm, lies the notion of separation of concern (Dijkstra, 1974) however, only subsequently could a consensus be found (Szyperski, Gruntz, & Murer, 2002) in which to characterise a software component as a unit of composition and deployment with contractually specified interfaces. Beyond the differences in the definition used in different models, sometimes specific for certain applications, the aim remains the same: increase the reusability and allow for the creation of new components from existing ones.

In general, a component can be seen as:

- A type of black box that encapsulates certain behaviours and has inputs and outputs ports, to communicate with other components in its environment.

- An independent unit having no context dependency; although as stated by Councill & Heineman (2001) a component may have an explicit context dependency on the operating system, a software component or some other software element.

- A unit that should be reusable for different applications. At a time where time to market should be kept to a minimum to increase competiveness, the creation of reusable artefacts is paramount.

- A unit of composition. It should be connectable to other components.
• A deployable unit. It should be deployable in different execution contest.

Moreover, Meyer (1999) defines some qualities that in his view make a good component:

• Careful specification: given a precise specification of component functionality and interface, the result of using a component should be well defined.
• Correctness: the result provided by a component should be predictable if used according to its specification.
• Robustness: when used properly, the component must not fail.
• Ease of identification: depending on their needs, users should be able to quickly and easily choose the appropriate component.
• Ease of learning.
• Generality: the component should be general enough to be used in different environments.

It is unrealistic to talk about software components without talking about the underlying component model. According to Councill & Heineman (2001), a component model is firstly a set of standards that define how to construct an individual component, and secondly it defines how components communicate and interact with each other. A number of component models exists from which we can cite, for example Sun’s Java Beans and Enterprise Java beans (EJB), Microsoft COM+, CORBA, UML 2.0, KobrA, CCM, and AADL (Lau & Wang, 2006). Each of these component models specifies how a component makes its services available to others and how components are composed with others to form larger components.

The notion of connecting components with others to form larger components is also known as composition. It is considered a central issue in component based engineering. Composition can take place at different stages of the life cycle of components. This cycle is composed of a design phase, where components are constructed and deposed in a component repository, a deployment phase, where components are extracted from a repository to create systems and finally a run-time phase where components are instantiated with data and ready to execute (Lau & Wang, 2006). The following figure (Figure 1) illustrates an idealised component life cycle.
Most of the component model previously mentioned relies on the notion of tightly coupled components. This means that components are one way or another dependant on other components. This has a negative impact on reusability. This research work attempts to tackle this issue, using a component model based approach based on exogenous connectors (Lau, Ling, & Wang, 2006).

### 2.2 Composition operators: Exogenous connectors

With exogenous connectors, the connectors control the flow of operations; they link components together calling methods and managing the results as required. The components responsibilities in this model are limited to computation. Components in this model are loosely coupled and therefore component behaviour might be more predictable (Lau, Elizondo, & Wang, 2005). In this component model, the components and connectors are built and deposited in a repository; they can then be retrieved and used to construct systems as required.
In exogenous connection, as in Figure 2, there are three types of entities:

- Firstly an atomic component which is composed of an invocation connector and a computation unit. The invocation connector is used to invoke methods in the computation unit. The computation unit encapsulates all the data and the computation of the component. In this component model, no control is initiated from the component.

- Secondly a composite component resulting from connecting a certain number of components with an exogenous connector.

- Thirdly an exogenous connector that encapsulates control. An exogenous connector is a control initiator; it is the entity that is used to compose components to form composite components.

An exogenous connector can be of different types from which the following are relevant for this work:

- When the exogenous connector is a pipe in a composite component, a method invocation for the composite component results in the pipe executing a method in one component, and using the result to initiate computation in another component.

- When the exogenous connector is a sequencer in a composite component, the sequencer invokes methods in different components sequentially.

- When the exogenous connector is a selector, it uses a condition to select the component whose methods need to be executed.
2.3 Elements of MATLAB and Simulink

2.3.1 MATLAB

MATLAB is the flagship product of ‘The Mathworks’ and it stands for MATrix LABoratory. MATLAB is a numerical computing environment that is geared towards the use of vectors and matrixes. It is an interpreted language where each line of a MATLAB program is read, interpreted and executed.

Application specifics add-on (Toolbox) developed by The Mathworks can be used to increased the functionality of the core of MATLAB, making MATLAB a tool of choice in many application domains such as signal processing, image processing, automatism, mechanical engineering, aerospace, automotive and many others.

2.3.2 Simulink

Simulink is another product of The Mathworks which is a type of add-on to MATLAB, widely used in system design. Simulink has been designed to numerically solve differential equations, linear as well as non linear, which underpin the behaviour of dynamic systems, represented by their mathematical models.

Simulink’s main attraction lies in its ability to work directly using block diagrams instead of using the mathematical equations, which underpin the system under consideration. This way of proceeding, is perfectly adequate, as it is how engineers usually work.

In an appropriate graphical environment, the user can easily assemble, by drag and drop, predefined block diagrams available from a library, to create a schematic representation of the system under construction.

Simulink allows the user to parameterise the block diagram and to organize them in a hierarchical form. The user can then switch from a higher level view of the system to a more detailed view of the system.

Once a model has been constructed in the graphical window and after the blocks have been parameterised, the simulation can start. Simulating is simply the process by which a system reacts to input signals to produce output signals. The software simulates the behavior of the model representing the system under study by calculating time step after
time step the value of the output signal giving the input signal and initial conditions. The following figure (Figure 3) represents an automotive application in Simulink.
3 Design considerations

3.1 Design using embedded MATLAB

As stated in the previous chapter, MATLAB plays a big role in many design and development workflows, specifically in the field of technical computing and research. Typically MATLAB is used in the start of algorithm design process. In many workflows, as pointed out by Zarrinkoud (2008), the development process starts in MATLAB and its power and flexibility is used to interactively design, visualize, verify and iterate the sophisticated algorithm that is being developed. This is true for various applications but in particular in application domains such as automotive or avionics domains, where The Mathworks tools are widely used. However, as different phases of the development are traversed, such as research, prototyping, simulation, design and finally the implementation, it is the job of many programmers and engineers to take the initial ideas expressed in the MATLAB language, and translate those ideas into languages that are more suitable for the final implementation on embedded systems. Those translations are usually performed manually, making it necessary to maintain multiple copies of the same algorithm when transferring from one phase to another. This is not only error-prone but also very time consuming. One solution developed by The Mathworks to tackle those issues is the embedded MATLAB subset.

3.1.1 What is embedded MATLAB

Embedded MATLAB is a subset of the MATLAB language that allows the generation of efficient embeddable C code directly from MATLAB code, usually for deployment in embedded systems. Embedded MATLAB supports over 270 operators and functions from the MATLAB language (Mathworks, 2009).
3.1.2 Advantages of design based on embedded MATLAB

Some of the benefits linked to embedded MATLAB include, but are not limited to the following (Mathworks, 2009):

- Provides seamless integration of algorithms designed using the MATLAB language in Simulink.
- Generates readable, efficient and embeddable C code from MATLAB code, reducing the risks of introducing errors, when manually translating the MATLAB code to a C code.

Although there are other advantages to using embedded MATLAB, the two advantages mentioned above are those that are the most relevant and significant. However, development based on embedded MATLAB comes with a set of limitations that the user needs to be aware of.

3.1.3 Constraints of design based on embedded MATLAB

As a subset of MATLAB, embedded MATLAB comes with a set of limitations associated with the features available in MATLAB. This can make it more demanding for someone used to the flexibility of MATLAB. As highlighted by The MathWorks (Mathworks, 2009), these limitations are essential to produce efficient C embeddable code.

- No support for cell arrays: whereas in MATLAB cell arrays are used to store different data types such as that illustrated below, embedded MATLAB only allow for arrays of a same data type.

\[
\begin{align*}
&>> c\{1\} = \text{uint8}(2); \\
&>> c\{2\} = \text{double}(3.5); \\
&>> c\{3\} = \text{’no EML’}; \\
&>> c \\
&c = \\
&[2] [3.5000] \text{’no EML’}
\end{align*}
\]

- Dynamic variables: while it is wholly acceptable to have the following two assignments in a MATLAB function, the size of all allocated memory need to be precisely known at the compilation time when in embedded MATLAB (EML)
Design considerations

>> s = [2, 3];
>> s = [4, 5, 6];

- **Global variables**: it is the case in MATLAB, that variables in a function are by default local, unless they are declared global; if this is the case, the variables are accessible elsewhere. In EML however it is not possible to declare a variable global. This can have a direct implication in the way variables are shared between functions; the only way to share the variables is by passing the variables as parameters to the function.

- **Java**: whereas MATLAB fully supports the Java language, (every installation of MATLAB comes with a Java Virtual Software (JVM) software) (Mathworks, 2009), EML does not support the Java language.

- **Matrix entry deletion**: in MATLAB it is perfectly acceptable to delete matrix entries, as the following example shows. However it is not acceptable to do this in EML since deleting an entry results in changing the variable size.

  >> M = [2 3 4; 3 5 6] %creating a matrix
  M =
      2   3   4
      3   5   6
  >> M(:, 1) = [] %deleting the first column the created matrix
  M =
      3   4
      5   6

- **Nested functions**: in MATLAB every function manages its own workspace; the scope of a variable is then limited to the function where the variable is declared. Using nested function however, allows a nested function to have access to the workspace of the enclosing function, in addition to its own workspace. EML does not allow for nested functions.

- **Sparse matrices**: in order to reduce the amount of memory to be allocated, MATLAB can used sparse matrices by squeezing out any zero elements. This is not supported in EML.

- **Object orientation**: EML does not support the object orientation features available in MATLAB.
• **Try/catch statement:** the error handling mechanism try/catch available in MATLAB or in any other OOP language is not available in EML.

### 3.2 Some limitations of Simulink

While the previous section focused on the limitations of embedded MATLAB, this section is more concerned with the higher level limitation of Simulink, as this may have an incidence on any design decision to be made.

- Primarily, there is no way to pass dynamically sized arrays between Simulink blocks in the version of Simulink used in this research work. Namely, the size of input parameters to a Simulink block or output parameters from a Simulink block cannot be changed at run time.

- Secondly, Simulink does not handle strings as data type for signals.

To address some of the shortcomings of Simulink, The MathWorks introduced in the latest version of Simulink, the possibility to handle C-style data structures.

### 3.3 Signals, buses and structures in a Simulink environment

A signal is defined in Simulink as a time varying quantity that has value at all time. Mathematically expressed, a signal is the quantity $s(t)$, $\forall t \in [t_0, t_1]$ with $t_0$ the simulation start time and $t_1$ the simulation stop time. As highlighted by Mathworks (2009), Simulink signals are simply mathematical, not physical entities and consequently, lines connecting Simulink blocks solely represent mathematical relationships amongst the signals defined by the block diagram.

In Simulink, a bus is simply a signal composed of other signals that can have different data types and sizes. Buses are either virtual or non virtual with virtual buses existing only as graphical conveniences. Throughout this work, the term bus is simply used to reference a non virtual bus, which contrary to a virtual bus has some functional effects and appears in generated code as data structure.

When using embedded MATLAB in a Simulink environment, the only way an EML interacts with a Simulink bus is through a data structure.
3.4 The concept of library in Simulink

One of the core features of Simulink is the existence of block libraries. A library in Simulink is a set of blocks that can be employed by users when creating other libraries or models. A new model is usually created by dragging and dropping pre-existing blocks into the model canvas. A block created from a library block is known as a reference block. Simulink then creates a link between the library block and the reference block in the model, so that any changes to a library block propagates to any of its instance blocks in a model. This concept of library is very useful, especially as it is possible to create a custom library, promoting the concept of reusability.
4 Design and implementation

4.1 General requirements on the design of component

Generally speaking, components usually publish their interfaces. This allows the user of a component to know what services the component provides and what services it requires. The following lists some of the main requirements that have been depicted from the semantics of component and that have guided the design.

- The design of component or composition operators should provide a way to publish a component interface. In language such as Java, facilities akin to reflection and introspection exist to examine the internal structure of any class at run time (Sagar, 1998). Such facilities do not exist in MATLAB, let alone embedded MATLAB.

- A component system should be self-similar, meaning that every composition can also be used as a component in the same system (Armstrong, Allan, Bernholdt, & Elwasif, 2005). To be able to achieve this, any component, atomic or composite component should externally look the same. The following figure (Figure 4) conceptually shows a component (atomic or composite).

![Figure 4: Schematic of a Component showing inputs and outputs ports](image)
4.2 Definition of input structure

As a component encapsulates some behaviour through different methods, the natural way to communicate with a component would be to provide the component with the name of the method which needs to be executed, plus the parameter values needed by the method, is shown in the following figure:

![Component with the method name and inputs parameters](image)

Figure 5: Component with the method name and inputs parameters

However as methods in a same component may not only have different signatures but require different numbers of input parameters, an implementation following Figure 5, would require a variable number of input parameters, depending on the method that is warranted. Simulink and henceforth embedded MATLAB, as stated already, do not allow for variable sized parameters. The method name as depicted in Figure 5 cannot be a string data type, as string data types are not supported in Simulink/EML.

To adhere to the self-similarity requirement stated previously, the method name and inputs parameters are combined together in a single structure, where the sizes and types of the fields in the structure are known at compilation time.

The following lists the fields of the input structure, along with a description of the purpose of the field:

- **descriptionFlag**: This boolean flag is used as a way to inform the component to either provide a description of its interfaces or to perform a computation.

- **numberOfMethods**: This unsigned integer provides information about the number of methods in the input structure. This is always set to 1 and is obsolete in the present version.

- **methodName**: Due to the fact that Simulink and consequently EML do not support string, the name of the method is converted to a unsigned 8 bits integer.
array. However as the length of the converted method name may vary, it is essential to allocate a constant maximum size and only use a subpart of the constant-size buffers.

- `inBusInfo.numberOfInputs`: This field provides valuable information about the number of inputs the method requires. This is principally important when it comes to determining the size of the valid sub-portion of the constant size buffer, pre-allocated for the inputs types and values.

- `inBusInfo.inputsTypes`: By default Simulink models use the default data type which is “double” and treat any values as a double quantity. Given that a piece of code that is intended for an embedded system causes the memory to be very limited, the use of more appropriate data types is usually required. This field is used to inform how a component will interpret the corresponding inputs values. The following table which assigns to each data type a corresponding unsigned 8 bits integer is used for this purpose (Table 1).

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Uint8 Value</th>
<th>Data Type</th>
<th>Uint8 Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>noType</td>
<td>0</td>
<td>Uint32</td>
<td>5</td>
</tr>
<tr>
<td>Boolean</td>
<td>1</td>
<td>Int8</td>
<td>6</td>
</tr>
<tr>
<td>Single</td>
<td>2</td>
<td>Int16</td>
<td>7</td>
</tr>
<tr>
<td>Uint8</td>
<td>3</td>
<td>Int32</td>
<td>8</td>
</tr>
<tr>
<td>Uint16</td>
<td>4</td>
<td>Double</td>
<td>9</td>
</tr>
</tbody>
</table>

*Table 1: Data Type-Uint8 correspondence*

- `inBusInfo.inputsValues`: This field is used to gather the input values to the method. The values are all treated as double quantity simply because data types cannot be mixed in an array. The component uses the information on the type save in the previous field, to downcast the value to the actual data type before computation.

The following example illustrates how some fields of the inputs structure are used:

Example 1: Suppose we are dealing with a function were a C – style signature is
double deposit (uint16 account, double amount) with one single output of type double and two inputs respectively of type uint16 and double. The following figure (Figure 6) shows how the representation of this function would look internally for the input structure

![Figure 6: Internal representation of input parameters and method name](image.png)

In the above example the method name is represented internally as the unsigned 8 bits integer, a representation of the string ‘deposit’. The input values are all double quantity but the information provided by the input types is used in the component to format the values to the right types. A description of the different fields in the input structure is summarized in the following table (Table 2) and a full listing of the method used to define the input structure is given in Appendix F: Input and output structure definition.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>descriptionFlag</td>
<td>boolean</td>
<td>1</td>
</tr>
<tr>
<td>numberOfMethods</td>
<td>uint8</td>
<td>1</td>
</tr>
<tr>
<td>methodName</td>
<td>uint8</td>
<td>[1 4000]</td>
</tr>
<tr>
<td>numberOfInputs</td>
<td>uint8</td>
<td>1</td>
</tr>
<tr>
<td>inputsTypes</td>
<td>uint8</td>
<td>[1 100]</td>
</tr>
<tr>
<td>inputsValues</td>
<td>double</td>
<td>[1 100]</td>
</tr>
</tbody>
</table>

Table 2: Inputs structure field description
4.3 Definition of output structure

As an initial approach, the need of a component to not only provide computation results but also self-description, would lead to components requiring two output ports, one for the computation results and the other to provide its description. However, this would increase the cost due to managing two different output ports for each component. A unified output structure has been defined, that can also be used for description or for computation purposes. Again, to adhere to the self-similarity requirement, every component would have the same output structure.

The following lists the field of the output structure, along with a description of the purpose of the field:

- **isValidFlag**: This boolean value is used as a way to inform the calling entity whether the called entity is valid or not. An atomic component is always valid. However, a composite component can be valid or not valid, depending on whether or not the semantics of the composition has been adhere to when the composite component was created. To illustrate this, let us consider a composite component $C_3$ resulting from piping two components $C_1$ and $C_2$ as depicted in Figure 7. $isValidFlag$ is true if there exists at least one method in $C_1$ whose number of outputs and type corresponds unambiguously to the number of inputs and type of at least one method in $C_2$.

![Figure 7: $C_3 = pipe (C_1, C_2)$](image)

- **descriptionFlag**: This is used to inform the calling entity how to interpret the result, as a description of the component or as a result of computation.
• `errorBusInfo.errorCode` and `errorBusInfo.errorPath` are used to handle error and are explained in the next section on error handling.

• `numberOfMethods`: The use of this field is similar to that in the input structure, only that in the output structure this field takes the value 1 or more, depending on whether or not a computation or a description has been performed.

• `methodName`, `inputsTypes` and `inputsValues` are basically the same as in the input structure, the only difference being is their respective size. Instead of being vectors they are matrices to cater for the fact that when a description is being performed, a component usually has more than 1 method.

• `numberOfInputs`: The usage of this field is similar to its usage in the input structure, the only difference being that instead of being a scalar it is a vector. This is for the same reason why the `methodName` in the output structure is a matrix.

• `outBusInfo.numberOfOutputs`: This field refers to the number of outputs for every method specified in the `methodName` field.

• `outBusInfo.outputsTypes` and `outBusInfo.outputsValues` are given the same considerations that were given to the inputs information, as well as evidently taking into account the additional fact that they are also used for description purposes.

A description of the different fields in the output structure is summarized in the following table (Table 3):

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Size</th>
<th>Field Name</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>isValidFlag</td>
<td>Boolean</td>
<td>1</td>
<td>methodName</td>
<td>Uint8</td>
<td>[100 4000]</td>
</tr>
<tr>
<td>descriptionFlag</td>
<td>Boolean</td>
<td>1</td>
<td>numberOfInputs</td>
<td>Uint8</td>
<td>[100 1]</td>
</tr>
<tr>
<td>errorCode</td>
<td>Uint16</td>
<td>1</td>
<td>inputsTypes</td>
<td>Uint8</td>
<td>[100 100]</td>
</tr>
<tr>
<td>errorPath</td>
<td>Enum{left,right}</td>
<td>[100 1]</td>
<td>inputsValues</td>
<td>Double</td>
<td>[100 100]</td>
</tr>
<tr>
<td>numberOfMethods</td>
<td>Uint8</td>
<td>1</td>
<td>numberOfOutputs</td>
<td>Uint8</td>
<td>[100 1]</td>
</tr>
<tr>
<td>outputsTypes</td>
<td>Uint8</td>
<td>[100 100]</td>
<td>outputsValues</td>
<td>Double</td>
<td>[100 100]</td>
</tr>
</tbody>
</table>

Table 3: Outputs structure field description
A full listing of the method used to create the output structure is given in Appendix F: Input and output structure definition.

4.4 Error handling

As stated previously the EML does not offer any built in mechanism for error handling, making it necessary to manually build a mechanism to at least identify where an error has occurred. At this stage the focus is more on recognizing that an error has occurred rather than to effectively handle the error automatically. Two fields of the output structure are used for this purpose, the `errorBusInfo.errorCode` and the `errorBusInfo.errorPath`. The first is used to identify the type of error that has occurred. The value of this field can be set in a computation unit, for example when a computation error has occurred, or in a composition operator when a method name passed could not be found. The second is used to construct a path to the component that triggered the error. It is in principle similar to traversing a tree. The value of this field is set only in composition operators. The following example (Figure 8) illustrates this concept.

![Figure 8: Error path illustration](image)

In Figure 8, if an error occurred in `C2`, the field `errorBusInfo.errorCode` is set internally to a value depending on the type of error and therefore when the control is handed back to the calling operator, this composition operator can set the appropriate path value to the faulty component. This is repeated until the root operator for further consideration.
4.5 Atomic component

As mentioned previously, atomic components are entities where the computations for a given system are performed. It is normally composed of an invocation connector, for the control, and a computation unit whose sole purpose is to perform computation. For this work however, we have chosen to treat an atomic component as a single unit, where control and computation are in a certain way mixed together. Although this is a slightly different approach compared to the theoretical understanding of an atomic component, it is believed that the implication of this simplification is very limited locally and even more so as we are not considering the addition of externally developed methods.

An atomic component should, as stated already, be able to provide a description of the methods available for computation, as well as computation results. An atomic component is implemented using a Simulink block called a ‘Function-Call Subsystem block’ and an ‘embedded MATLAB function block’. A ‘Function-Call Subsystem block’ is used to represent a subsystem that can be invoked as a function of another block (Figure 9). It is usually executed by a Stateflow chart as a function-call initiator (Karris, 2008).

![Figure 9: Function Call Subsystem used to implement atomic component](image)

The following three figures (Figures 10-12) show snapshots of the code representing an atomic component for a bank account in a banking system. A full listing of this code is given in Appendix A: Atomic component source codes.
Design and implementation

```plaintext
#function y = fnLongComputationUnit(w)

% Input w
% type: componentInBus
% size: i
% Output p
% type: componentOutBus
% size: i
% @ 2009, Patrice Siatchoua

% This encapsulates a number of methods for a bank account
persistent balance;
persistent descStruct; % use internally to describe component
% of type componentOutBus

if isempty(balance)
  balance = double(100); % if not double type must be specified
end
if isempty(descStruct) % use for memory allocation
  ...
  descStruct.methodNames{i,1:length('balance')} = uint8('balance');
  descStruct.methodNames{i,1:length('deposit')} = uint8('deposit');
  descStruct.methodNames{i,1:length('withdraw')} = uint8('withdraw');
  ...
end
% Here a manual description of the methods need to be done
if (a.description-flag == true)
  y = descStruct;
else
  % first initialize the compStruct

  outputValues = double(error(1,100));
  for i = 1:n numberOfMethods
    ...
    for j = 1:descStruct.numberOfMethods
      ...
      % make sure that the method matches (method name, inputs number and inputs types)
      if (all(strcmp(methodNames == methodNames) == true)
        if (all(numberOfInputs == numberOfInputs)
          if (all(inputTypes == inputTypes) == true)
            % determine which function to call
            if (all(uint8('balance') == methodNames{i,1:length('balance')}) == true)
              result,errorCode = getBalance(double(balance));
            elseif (all(uint8('deposit') == methodNames{i,1:length('deposit')}) == true)
              ...
            elseif (all(uint8('withdraw') == methodNames{i,1:length('withdraw')}) == true)
              ...
            else
              errorCode = 1;
              end
              % fill in the results
              ...
          if (errorCode == 0) % an error has occurred, exit
            y = compStruct;
            return;
          end
          end
        end
      end
    end
  end
```

Figure 10: Atomic component for a bank account (part 1)

Figure 11: Atomic component for a bank account (part 2)
In the bank account illustrated above, the following operations are available:

- A possibility to make a deposit using the function deposit
- A possibility to make a withdrawal using the function withdraw
- A possibility to query the balance using the function getBalance

These three functions are elements of the computation unit; it is where the computations are performed for this atomic component. They are implemented as subfunctions (line 69 to line 90 of Figure 12) of a primary embedded MATLAB function.

In any atomic component, a description of the interface of the component should be provided through the output structure `descrStruct` defined in line 13 of Figure 10, in the case of a bank account atomic component. It is declared to be persistent and to mean that the variable is local to the function in which it is declared, yet its value is retained in memory between calls to the function (Mathworks, 2009). The fields of the output structure `descrStruct` are filled in from line 19 of Figure 10, according to the definition of output structure provided in 4.3.

The control part of the atomic component is provided in the case of the bank account in Figure 11. Thereby, depending on the name of the method present in the input structure,
the requested operation is performed and the results are gathered to produce the output structure.

This approach of mixing computation and control in one single entity, results in having to implement the methods of the computation units as subfunctions. This has at least two implications. Firstly, the only way to deal with parameter types is to undertake it in the way it is assumed in the function *deposit* in lines 74 and 75 of Figure 12. Secondly, as the subfunctions do not have direct access to variables declared in a primary function, those variables need to be passed as parameters of subfunctions, in case a subfunction needs to access the value of those variables. Take for instance the function *deposit* defined in Figure 12, in the output structure, after the description process, this function would appear to have only one single parameter, but internally, as it needs to access the persistent variable *balance* declared in the primary function, it has to have two parameters.

The atomic component presented here will be the building blocks of composite components presented in a later section and also when the different composition operators need to be tested.

### 4.6 Composition operators

The composition operators or exogenous connectors implemented during this work are the pipe, the sequencer and the selector. Given the fact that composition operators or simply operators described in the component model implemented in this work, are in a certain sense control operators (control always start from one of these operators), a Simulink building block that was chosen to implement the different operators was the Stateflow chart block. The next sections present the detailed implementation of the different operators based on Stateflow.

#### 4.6.1 Pipe connector

Let us recall the semantic of a pipe. Given two components *A* and *B*, and a pipe that connects these two components, the resulting system is only valid if there exists at least one function in *A*, whose number of outputs and type correspond to the inputs and types of a function in *B*. The operation of a pipe is similar to the composition operation ° between two mathematical functions *f* and *g* denoted *f*g*, where *f* is a function whose
domain includes the range of $g$. The top level view of the pipe is illustrated in Figure 13, along with the name of the inputs and outputs ports.

![Pipe connector using a Simulink subsystem block and a Stateflow chart](image)

*Figure 13: Pipe connector using a Simulink subsystem block and a Stateflow chart*

In Figure 13, there are three input and five output ports, the top two most ports `mainDataIn` and `mainDataOut` that correspond respectively to the Stateflow chart (port `dataIn` and `dataOut`) are used to receive input information from higher level entities, and to send back output information to the higher level entity that has requested an operation. The others ports are used to connect to lower level components (atomic or composite). The best way to understand how the different ports are used is through an illustration. At this point we are only focusing on explaining the use of the different ports of the pipe. Figure 14 shows a pipe connected to two atomic components as described in the previous section. From left to right:

- `rhsDataOutput` is used to receive an output structures from the component connected to the right of the pipe.
- `lhsDataOutput` is used to receive an output structure from the component connected to the left of the pipe.
- `rhsDataInput` is used to send data (input structure) to the right component after processing the data, if required.
- **lhsDataInput** is used to send pre-processed data to the left component.

- **rhsInvoke** is an event port used to trigger the execution of the right component. The atomic components are implemented as a Simulink function-Call subsystem and they execute only if triggered by a function-Call initiator; in this case, if the event **rhsInvoke** is present.

- **lhsInvoke** is similar to **rhsInvoke** but for the left component.

![Figure 14: Pipe connector with two atomic components](image)

It is worth recalling that apart from the two events ports **rhsInvoke** and **lhsInvoke**, all the ports of the pipe are inputs or outputs structures, as defined previously. The type of the port is determined by looking at how this port is linked to the component. If the port is connected to the inport of the component, it is an input structure; otherwise it is an output structure. Having looked at the top level representation of the pipe and its connections port, the internal structure of the pipe will be analysed.
Any connector is internally represented by a number of states to perform its operation. When the connector is active it enters the transition, through the default transition, to enter the control state. Upon entering the control state, it unconditionally goes to state0 where it will transit to a description state or a computation state, depending on what the calling entity has requested. Again, the request of the calling entity is coded in the descriptionFlag field of the input structure. Figure 15 illustrates the control state and its different substates. In Figure 15, additional boolean variables (descriptionDone and computingDone) are used to avoid infinite loop and to ensure that after a single description or computation operation is performed, the control returns to state0 and does not transit to another state, unless a new operation is requested from an entity above in the system hierarchy.
The description state is used to control the description operation. This state works by initially requesting the left component to give a description of the methods it contains (transition state0-state1 of Figure 16). After receiving the result of the description operation for the left component, it requests the same operation for the right component (transition state1-state2). In state2 the results of the two operations that are available through \( lhsOutputIn \) for the left component and \( rhsOutputIn \) for the right component are gathered together in a single output structure. This will eventually be transferred back to the calling entity. The way the gathering works is always specific to the type of connector that is being considered. For the pipe, the gathering performs by creating new methods from processes in the left and right components that satisfy the semantics of the pipe. Figure 17, illustrates a system composed of two components having three methods and having two methods, . Assuming that for method the number and type of outputs corresponds to method \( s \) number and type of inputs, then state2 of Figure 16 will create a new function , whose input information (number and type) correspond to those of and whose output information (number and type) correspond to those of .

*Figure 16: Description state*

*Figure 17: Pipe methods matching*
It is clear from Figure 15, that computation is requested when the `descriptionFlag` field of the input structure is set to the Boolean false value. When in state computation, the pipe connector goes through a certain number of substates described below, to perform its operations (Figure 18).

The `Initial_Description` state’s purpose is to provide a description of the methods available in the composite component, formed by the connector and the connected components. Internally, this state is similar to the description state of Figure 16, the only difference being that the `descriptionFlag` field value needs to be set to false explicitly, as it is true when the computation state is triggered. At the end of the `Initial_Description` state, new method names are created, as described previously in Figure 17.

The `checkAvailability` state’s function is to compare the name of the method in the input structure, in addition to the input information (number of inputs, type of inputs), to the list of methods that resulted from the previous state. At the end of this state, either a matching method is found or an output structure is generated with the `errorCode` field set to a value greater than 0.

In the case where a matching method is found in the previous state, the connector moves to the `extraction` state where it creates two methods with the corresponding set of input information for the connected components. A simple way to illustrate this is by
considering the small system in Figure 17. Ideally the input structure would have a method named and the connector would extract method for the left component and method for the right component, evidently along with the corresponding input parameters.

The state where the pipe connector controls the computation is the computing state, whose substates are shown in Figure 19. Having extracted the two methods required by the connected components, it then effectively requests the component to perform the required computations. This is done by first invoking the left component to do the computation (in state leftComputing) and then the right component (in state rightComputing), using the result of the first computation. If an error occurs during the computation, the errorPath field of the output structure is updated as described in section 4.4.

![Diagram](image)

*Figure 19: Computing state of the pipe connector*

The full listing of the codes used to implement the pipe connector is given in Appendix B: Pipe connector source codes.

### 4.6.2 Sequencer connector

Given two components and are connected through a sequencer connector, the resulting system is always a valid one. In other words, there are no restrictions on how two components are connected using a sequencer. It is only a matter of performing one
operation from the first component, followed by a second operation from the second component. The top level view of the sequencer connector, seen in Figure 20 is similar to that of the pipe connectors, in terms of the ports both used. The purpose of these ports is similar to that of the pipe, described previously. Even the internal structure of both connectors is very similar, at least to the point where the different operations that are part of the state transition diagram have the same higher level purpose. The difference is regarding the implementation, as clearly the two connectors’ purposes are different.

The sequencer connector initiates in a \textit{control} state, in the same way as the pipe connector, and transitions to either the description or the computation state, depending on the requested operation from the higher level entity.

The \textit{description} state is internally similar to that of the pipe and follows the same structure, illustrated in Figure 16. The only difference is the way the new methods are created. Consider a system, as illustrated in Figure 21, with a sequencer connecting two components $C_0$ with methods $A, B, C$ and $C_1$ with methods $A', B'$. The description of this system results in a list of methods ensuing from the Cartesian product $\{A, B, C\} \times \{A', B'\}$ with $\times$ the Cartesian product operator. The inputs of the resulting methods are formed by combining the inputs of the constituents’ methods; the same applies for the outputs.
The computation state structure follows exactly that of the pipe with an Initial_Description state, where the methods available for computation are listed as in Figure 21. A checkAvailability state where the connector compares the method and parameters’ names in the input structure, to those resulting from the Initial_Description state. This is then followed, if successful, by the state extraction, where the sequencer extracts method information for the connected components to be processed in the computing state.

### 4.6.3 Selector connector

As with the sequencer connector, a system resulting from connecting two valid components with a selector is always valid. The purpose of the selector is to select one component based on a certain condition, to execute a certain method.

In contrast to the previously described connector, a Simulink block mask was used to proficiently deal with the condition expression. A mask is a custom user interface that can be used to hide a subsystem content (Mathworks, 2009). The benefit of the block mask is its parameter dialog box. This has been customised to allow for conditional operator and condition values to be selected or entered (Figure 22).
In terms of the approach used, the implementation of the selector is, in a certain way similar to that of the previously described connectors.

The description state of the selector differs from the other connectors’ description state in the way the new methods are created. For illustration purposes let us consider the system depicted in Figure 23. The system is composed of two components \( C_0 \) with methods \( A, B, C \), another way to look at this, is to consider the \( C_0 \) and \( C_1 \) as the following two sets: \( C_0 = \{ A, B, C \} \), \( C_1 = \{ A, B, D' \} \), requesting a description for this system simply results in the union of these two sets \( C_0 \cup C_1 = \{ A, B, C, D' \} \). The methods that are common to both components are then augmented with an additional parameter; this is later used as the condition variable which is necessary to choose between the two components.
The *computation* state structure follows the same structure as the previously described connectors, namely *Initial_Description*, *checkAvailability*, *extraction* and *computing* states. However, in contrast to what was the case for the pipe connector, the *extraction* state is mainly concerned with evaluating the condition expression and deciding which component needs to be selected for the computation. The *computing* substate, effectively uses the information to invoke the appropriate component (Figure 24).

![Figure 24: Computing sub state for a selector](image)

### 4.7 Component repository

The component repository provides the foundation, on which the reusability of components is based. This is where the Simulink library concept comes to the fore. It provides a simple way to build a component repository from reusable artefacts. Figure 25 shows a snapshot of the component repository implemented during this research work. It is composed of the different exogenous connectors, in addition to atomic components and a composite component.
Design and implementation

Figure 25: Component repository
5 Testing

This section is chiefly concerned with testing the implemented components model elements. In order to perform the test, different systems, as shown in Figure 27, are built in the Simulink environment from pre-existing components, taken from the component repository of Figure 25. As Simulink has a way to access MATLAB workspace, input parameters are fed to the built system through MATLAB configuration files. In the same way, outputs values are processed using a MATLAB function, to produce a human readable output in the form of a simple file. The reason for using MATLAB is primarily for convenience. Figure 26 shows the test environment used to perform the tests.

![Figure 26: Test environment](image)

To validate the model built in Simulink many tests have been performed from which a subset configuration is shown in Figure 27.

The first group of tests (1 and 2), is concerned with testing an atomic component, which in this case represents a simple bank account as illustrated in Figure 10 to Figure 12.

The second group of tests (3, 4, 5, 6, 7 and 8) is concerned with testing a 1st level composite component using the pipe connector for test 3 and 4. For test 5 and 6 the sequencer connector is used and finally for test 7 and 8 the selector connector is employed. The tests are performed using the atomic component from the first group of tests.
The third and last groups of tests (9, 10, 11 and 12) represent a 2nd level composite component with one connector utilised to compose a composite component and an atomic one. The tests are performed using two pipe connectors (test 9 and 10) and a combination of pipe and sequencer connectors (test 11 and 12).

For all the tests, two operations are performed. Firstly a description is requested, to establish the available methods that can be called and then a computation is performed, using one of the methods discovered during the description process.

### 5.1 MATLAB test files

Principally two MATLAB files are used during the tests; the first one shown in Figure 28 is used to assign values to the input structure of the system under test. The variable `descriptionFlag` is altered depending on the test being performed; it takes the value `false` if a computation is needed or `true` if a description is needed. The second file is used to process the output of the simulation and is provided in Appendix G: MATLAB processing file.
5.2 Configuring and running the simulation

The first thing required to specify within a simulation is the length of running time. This is done by providing the start and stop time. For the purpose of the different tests performed, both values were kept to the same value of 0 to ensure that the simulation runs only a once.

The next thing to specify is the solver. The solver is used by Simulink to compute the inputs, outputs and states at intervals, between the specified start and stop times (Mathworks, 2009). The solver that has been used during the test is the fixed-step discrete solver. According to Mathworks (2009), this solver is suitable for systems or models that have no continuous state, including models that have only discrete states. This is the case for all models within this research.

```matlab
1 % configuration script to test the component model
2 % (c) Patrice Siatchoua, 2009
3 clc
4 clear all
5 load newVersionComponentBusen.mat; % load the bus needed
6 descriptionFlag = false;
7 numberOfMethods = 1; \always 1
8 methodNames = uint8(size(1,4000));
9 % using the following format add an empty function as needed
10 methodNames(2) = length('getBalance'|| 'uint8' || 'getBalance');
11 numberOfInputs = uint8(1);
12 numberOfInputs(2) = 1;
13 inputsTypes = uint8(size(1,100));
14 inputsTypes(1) = 9;
15 inputsTypes(4,1) = 0;
16 inputsValues = double(zeros(1,100));
17 inputsValues(1) = 1;
18 inputsValues(5,1) = 322.34;
```

Figure 28: MATLAB configuration file

Figure 29: Simulink configuration parameters
Another configuration found to be very useful, was to facilitate the super step semantics (see Figure 30) in any of the models having state charts. This is specifically the case for any composite component. The super step semantics allows the Stateflow chart to execute multiple times, for every time step when the chart has no inputs events (Mathworks, 2009).

One last configuration that was made before running the simulation was to alter the model properties callbacks of the system under test, to include the two previously mentioned MATLAB files, of section 5.1, as part of the model initialisation and simulation stop function. This ensured for instance, that the buses elements were loaded in the MATLAB workspace (Figure 28: line 28) before the start of the simulation and that the results were automatically processed, to produce the human readable outputs at the end of the simulation.

5.3 Analysing the results

5.3.1 Atomic component

The results of performing the tests 1 and 2, laid out in Figure 27, for an atomic component representing a simplified bank account are depicted in Figure 31. In the left hand side of this figure are the processed results obtained by requesting the component to publish the methods it contains, in other terms its description. This is again emphasised in the results due to the descriptionFlag value being true. The available methods, in addition to inputs and outputs information, are parts of the results. From the

![Figure 30: Simulink configuration parameters- chart options](image)
interface information obtained, requesting a computation with one of the methods, in this case ‘getBalance’, results in what is shown in the right hand side of the figure.

![Diagram](image)

**Figure 31: Result of testing an atomic component**

### 5.3.2 Composite component using a pipe connector

The results of performing the tests 3 and 4 for a composite component, using a pipe connector and two atomic components are depicted in Figure 32. Here the atomic components are both references of a bank account atomic component, as tested in the previous section. Again the left hand side of the figure shows the results of requesting a description and the right hand side the results of requesting a computation; in this case the computation is requested for the method ‘withdrawdeposit’. To better understand the resulting output value, it is worth pointing out that the method ‘withdraw’ in the atomic component, outputs the new balance after a withdrawal operation has been performed. The two atomic components each have a balance of 100 to start with. Calling the method ‘withdrawdeposit’ with the inputs value of 20 results in the left component performing a withdrawal and the result is piped through to the second component, to perform a deposit which results in outputting the new balance.

One thing worth pointing out is the nature of the method names. At this stage composite names for method names were chosen when dealing with composite components. It might not be the most effective way to hide complexity but at this stage it helps follow what is happening inside the components. Improvements related to the better choice of method names have been kept for future work.
### 5.3.3 Composite component using a sequencer connector

The results of performing the tests 5 and 6 for a composite component, using a sequencer connector and two atomic components are depicted in Figure 33. Again the atomic components are references of the bank account atomic component. However, the only difference is that the left component has a balance of 100 and the second has a balance of 200 to start with. Within this test the left hand side shows only part of the results following a description request and the right hand side shows results from requesting the computation of the method ‘getBalanceddeposit’ with the inputs 100. This will effectively output the balance from the left component, as well as output the new balance of the right component after the deposit.
5.3.4 Composite component using a selector connector

The results of performing tests 7 and 8 for a composite component, using a selector connector and two atomic components are depicted in Figure 34. Investigating the number of inputs for each of the methods from the left hand side of the figure, it can be seen that the number of inputs is higher compared to Figure 31. Recalling what was said in section 4.6.3, this is simply due to the fact that the common methods associated with the two components are augmented with a condition variable.

The right hand side shows the result of requesting the computation of the method ‘deposit’ after a similar condition to Figure 22. This is evaluated to true and is used to choose the second component.

5.3.5 Composite component using two pipe connectors

Figure 35 shows the results of performing the tests 9 and 10 for a two level composite component using two pipes. The left hand side of the figure shows the result following a description request, whereas the right hand side shows the results after requesting the computation of the method ‘withdrawdepositdeposit’. All three atomic components used in this test are references of the bank account atomic component tested previously, with an initial balance of 100. The result is obtained by first performing a computation of the subsystem identical to the subsystem used in Figure 32. Subsequently the result is then piped through to the next atomic component to obtain the final result of 280.
5.3.6 Composite component using a pipe and a sequencer connectors

The next figure (Figure 36) shows the results obtained by performing the tests 11 and 12 for a two level composite component, using a pipe connector for the innermost component and a sequencer connector as a root connector. Again, the three atomic components are references of the bank account atomic component with initial values of 100. The left hand side as for the previous tests represents the results obtained after requesting a description, whereas requesting a computation for the method ‘withdrawdepositgetBalance’ results in the detail in the right hand side of the figure. Following the previous test, the left component is evaluated and instead of piping the result to the next component, the next component is evaluated independently to produce the required outputs.

```
ValidFlag: True  ValidFlag: True
errorCode: 0     errorCode: 0
errorPath: none  errorPath: none
descriptionFlag: True  descriptionFlag: False
method name(s): getBalance() and deposit()  method name(s): getBalance() and deposit() and withdraw()
withdraw() and deposit()  number of inputs for each method:
number of outputs for each method:
number of inputs for each method:
number of outputs for each method:
number of inputs for each method:
number of outputs for each method:
number of inputs for each method:
number of outputs for each method:
```

Figure 35: Result of testing a composite component using two pipes

Figure 36: Result of testing a composite component using a pipe and a sequencer
6 Conclusion

6.1 Summary of achievements and critical analysis

This work aimed to develop a suitable methodology to use the component based software engineering concept with Simulink. The component model based on the use of exogenous connectors was chosen as the basis of this investigation. The approach that was proposed enabled components (atomic or composite) to publish their interface. This is an area we believe to be paramount within the proper utilisation of components. Furthermore, we have shown that by appropriately choosing pre-existing Simulink artefacts, such as the function-call subsystem, along with carefully designed Stateflow charts in conjunction with the embedded MATLAB subset, it is possible to implement most of the exogenous connectors that form the kernel of the corresponding component model. Furthermore, the availability of a library concept in Simulink has facilitated the development of component repository, which is at the heart of component reusability and hence very important for any component based model. The results of the tests performed, agreed with what was expected from the semantics of the different connectors used. Nevertheless, some restrictions linked to the development of components based systems using this approach, remain to be addressed:

- First and probably the most prominent restriction is related to variable sized array. In Simulink it not possible to use variable size array, consequently embedded MATLAB requires that the size of all variables be known at time of compilation. This has some implications on the concept of reusability. For a certain system, it is possible by analysing the system correctly to discover the components that the system requires and fix the size of some variables. In the approach used in this research work, Simulink buses or structures with fields with known size and type were used and shared among all the components. If all the components are developed for a bespoke system, this approach is believed to work well. However if the components are developed mainly from a domain
analysis, it is implausible to know beforehand which system will be built from the components and consequently the size of the fields of the structure might not be known at compile time.

- A second point, albeit less prominent is concerned with complexity hiding. The name of methods from composite components complexity was not properly hidden, making it possible to guess from the name of the methods, the structure of the components. The self similarity principle would require that the component look the same, at least from the way the component is named. It should not be possible from the outside to make a distinction between an atomic component and a composite one.

### 6.2 Future works

This work was only a first step towards using Simulink for component based software development, based on exogenous connectors. Although good results have been achieved, some further work requires to be undertaken in the future. For that we intend to

- Use larger examples to prove that our approach is fit for purpose. We plan to use a missile guidance system; such a system already exists as a Simulink model, it would be interesting to compare both approaches.

- Address the second point outlined in the previous section by creating new names for methods when it comes to composite components, so that it is not possible to differentiate to atomic component.

- Address the problem of how to develop components resulting from a domain analysis, giving the restriction of embedded MATLAB. Here we would look at alternative ways to obtain the same result by investigating other MATLAB tools.
function y = fcnLhsComputationUnit(u)
%#eml
% Input: u,
% type: componentInBus
% size: 1
% Output:y
% type: componentOutBus
% size: 1
% (c) 2009, Patrice Siatchoua

% This encapsulated a number of methods for a bank branch
persistent balance;
persistent descrStruct; % use internally to decribe component
tof type componentOutBus

if (isempty(balance))
balance = double(100); % if not double type must be specified
end
if (isempty(descrStruct)) % use for memory allocation
    descrStruct isValidFlag = true;
descrStruct.setDescriptionFlag = u.descriptionFlag;
descrStruct.errorBusInfo.errorCode = uint16(0);
descrStruct.errorBusInfo.errorPath = EnumPathValues(zeros(1,1));
descrStruct.numberOfMethods = uint8(3);
descrStruct.methodName = uint8(zeros(1,100));
descrStruct.methodName(1,1:length('getBalance')) = uint8('getBalance');
descrStruct.methodName(2,1:length('deposit')) = uint8('deposit');
descrStruct.methodName(3,1:length('withdraw')) = uint8('withdraw');
descrStruct.numberOfInputs = uint8(zeros(100,1));
descrStruct.numberOfInputs(1) = uint8(0);
descrStruct.numberOfInputs(2) = uint8(1);
descrStruct.numberOfInputs(3) = uint8(1);
descrStruct.inputsTypes = uint8(zeros(100,100));
descrStruct.inputsTypes(1,1) = uint8(0);
descrStruct.inputsTypes(2,1) = uint8(9);
descrStruct.numberOfOutputs = uint8(zeros(100,1));
descrStruct.numberOfOutputs(1) = uint8(1);
descrStruct.numberOfOutputs(2) = uint8(1);
descrStruct.numberOfOutputs(3) = uint8(1);
descrStruct.outputsTypes = uint8(zeros(100,100));
descrStruct.outputsTypes(1,1) = uint8(9);
descrStruct.outputsTypes(2,1) = uint8(9);
descrStruct.outputsTypes(3,1) = uint8(9);
descrStruct.numberOfOutputsValues = double(zeros(100,100));
descrStruct.outBusInfo.numberOfOutputs = uint8(zeros(100,1));
descrStruct.outBusInfo.numberOfOutputs(1) = uint8(1);
descrStruct.outBusInfo.numberOfOutputs(2) = uint8(1);
descrStruct.outBusInfo.numberOfOutputs(3) = uint8(1);
descrStruct.outBusInfo.outputTypes = uint8(zeros(100,100));
descrStruct.outBusInfo.outputTypes(1,1) = uint8(9);
descrStruct.outBusInfo.outputTypes(2,1) = uint8(9);
descrStruct.outBusInfo.outputTypes(3,1) = uint8(9);
descrStruct.outBusInfo.outputValues = double(zeros(100,100));
end

% here a manual description of the methods need to be done
if (u.descriptionFlag == true)
y = descrStruct;
else
% the computation happen here
% first initialize the compStruct
compStruct.isValidFlag = true;
compStruct.descriptionFlag = false;
compStruct.errorBusInfo.errorCode = uint16(1);
compStruct.errorBusInfo.errorPath = EnumPathValues(zeros(100,1));
compStruct.numberOfMethods = uint8(0);
compStruct.methodName = uint8(zeros(100,100));
compStruct.numberOfInputs = uint8(zeros(100,1));
Appendix A: Atomic component source codes

```matlab
compStruct.inputsTypes = uint8(zeros(100,100));
compStruct.inputsValues = double(zeros(100,100));
compStruct.outBusInfo.numberOfOutputs = uint8(zeros(100,1));
compStruct.outBusInfo.outputsTypes = uint8(zeros(100,100));
compStruct.outBusInfo.outputsValues = double(zeros(100,100));

nberMethodCount = 0;
outputsValues = double(zeros(1,100));
for i = 1:u.numberOfMethods % ideally it should only be one single calculation in a
  computation unit at any single time
    uMethodNames = u.methodName(i,:);
    uNumberOfInputs = u.inBusInfo.numberOfInputs(i);
    uInputsTypes = u.inBusInfo.inputsTypes(i,:);
    uInputsValues = u.inBusInfo.inputsValues(i,:);
    for j = 1:descrStruct.numberOfMethods
      methodNames = descrStruct.methodName(j,:);
      numberOfInputs = descrStruct.numberOfInputs(j);
      inputsTypes = descrStruct.inputsTypes(j,:);
      % make sure that the method matches (method name, inputs number and inputs
types)
      if (all(uMethodNames == methodNames) == 1)
        if (uNumberOfInputs == numberOfInputs)
          if (all(uInputsTypes == inputsTypes) == 1)
            % determine which function to effectively call
            if (all(uint8('getBalance') ==
              methodNames(1:length('getBalance'))) == 1)
              [result,errorCode] = getBalance(double(balance));
              outputsValues(1) = double(result);
            elseif (all(uint8('deposit') == methodNames(1:length('deposit'))) == 1)
              [result,errorCode] = deposit(uInputsValues(1),double(balance));
              outputsValues(1) = double(result);
            elseif (all(uint8('withdraw') ==
              methodNames(1:length('withdraw'))) == 1)
              [result,errorCode] = withdraw(uint16(uInputsValues(1)),double(balance));
              outputsValues(1) = double(result);
            else
              errorCode = 1;
            end
            % fill in the results
            nberMethodCount = nberMethodCount + 1;
            compStruct.isValidFlag = true;
            compStruct.descriptionFlag = false;
            compStruct.errorBusInfo.errorCode = uint16(errorCode);
            compStruct.errorBusInfo.errorPath(1) = EnumPathValues.none;
            compStruct.numberOfMethods = uint8(nberMethodCount);
            compStruct.methodName(nberMethodCount,:) = methodNames;
            compStruct.numberOfInputs(nberMethodCount) = numberOfInputs;
            compStruct.inputsTypes(nberMethodCount,:) = inputsTypes;
            compStruct.inputsValues(nberMethodCount,:) = uInputsValues;
            compStruct.outBusInfo.numberOfOutputs(nberMethodCount) =
            descrStruct.outBusInfo.numberOfOutputs(j);
            compStruct.outBusInfo.outputsTypes(nberMethodCount,:) =
            descrStruct.outBusInfo.outputsTypes(j,:);
            compStruct.outBusInfo.outputsValues(nberMethodCount,:) =
            outputsValues;
        end
      end
    end
  end
end
```

Patrice Siatchoua – School of Computer Science - 2009
Appendix A: Atomic component source codes

% Subfunctions rely on strong typing. Type must be specified
% Upcasting or downcasting not allowed
% an error occured if there is a type mismatch
% error code for type mismatch start with 101:
% 10N representing type mismatch in the N methods

% get the balance
function [r, error] = getBalance(b)
% b is of type double and is local: not visible from outside
% r is of type double
% error is local and of default type
% from outside the function interface look like
% !! <double>r = getBalance() !!
% an xml description of this function could be
% <function>
%   <name>getBalance</name>
%   <inputNumber>0</inputNumber>
%   <outputNumber>1</outputNumber>
%   <outputType eq='9'>double</outputType>
% </function>
if isa(b,'float') == 1    %mandatory to check the type prior to any calculations
    r = b;
    error = 0;
else
    r = double(0);
    error = 101;
end

% make a deposit
function [r, error] = deposit(amount, b)
% amount is of type double
% b is of type double and is local
% r is of type double
% error is local of type double
% <function>
%   <name>deposit</name>
%   <inputNumber>1</inputNumber>
%   <inputType eq='9'>double</inputType>
%   <outputNumber>1</outputNumber>
%   <outputType eq='9'>double</outputType>
% </function>
if isa(amount,'float') == 1
    if isa(b,'float') == 1
        r = amount + b;
        error = 0;
    else
        r = double(b);
        error = 102;
    end
else
    r = double(b);
    error = 102;
end

% make a withdraw
function [r, error] = withdraw(amount, b)
% amount is of type uint16
% b is of type double and is local
% r is of type double
% error is local of type double
% <function>
%   <name>withdraw</name>
%   <inputNumber>1</inputNumber>
%   <inputType eq='4'>uint16</inputType>
%   <outputNumber>1</outputNumber>
%   <outputType eq='9'>double</outputType>
% </function>
if isa(amount,'uint16') == 1
    if isa(b,'float') == 1
        if double(amount) > b
            r = b;
        else
            r = double(b);
            error = 102;
        end
    else
        r = double(b);
        error = 102;
end
else
    r = b - double(amount);
end
error = 0;
else
    r = double(b);
    error = 103;
end
else
    r = double(b);
    error = 103;
end
Listing B1

function s = updateOutput(u1, u2)

% This function is used to assemble the description outputs of connected components into one single description structure.
% Inputs: u1 [type componentOutBus, size{1}] component connected left
% u2 [type componentOutBus, size{1}] component connected right
% Output: s [type componentOutBus, size{1}]
% Invariant: Structure fields must be assigned in the same order on all control flow paths.
% (c) 2009, Patrice Siatchoua

mismatchFound = false; % is there a matching function to be used for the pipe

dimMethodName = size(u1.methodName);
mismatchMethodName = uint8(zeros(1, dimMethodName(2)));
dimInputs = size(u1.outBusInfo.outputsTypes); % same for inputs and outputs
u1InputsTypes = uint8(zeros(1, dimInputs(2)));
u1InputsValues = double(zeros(1, dimInputs(2)));
u1OutputsTypes = uint8(zeros(1, dimInputs(2)));
u1OutputsValues = double(zeros(1, dimInputs(2)));
u2MethodName = uint8(zeros(1, dimMethodName(2)));
u2InputsTypes = uint8(zeros(1, dimInputs(2)));
u2InputsValues = double(zeros(1, dimInputs(2)));
u2OutputsTypes = uint8(zeros(1, dimInputs(2)));
u2OutputsValues = double(zeros(1, dimInputs(2)));

k = uint8(1); % used for the number of methods in the resulting structure

% initialisation of the structure s
s.isValidFlag = mismatchFound;
% set the description flag to true
s.descriptionFlag = true;
% update the errorBusInfo
s.errorBusInfo.errorCode = uint16(0);
theDim = size(u1.errorBusInfo.errorPath);
tp = EnumPathValues(zeros(theDim(1), 1));
s.errorBusInfo.errorPath = tp;
% update the number of methods
s.numberOfMethods = uint8(0);
% update the method name
s.methodName = uint8(zeros(dimMethodName(2)));
% update the number of inputs
s.numberOfInputs = uint8(zeros(dimMethodName(1), 1)));
% update the inputsTypes
s.inputsTypes = uint8(zeros(dimInputs(2)));
% update the inputsValues
s.inputsValues = double(zeros(dimInputs(2)));
% update the number of outputs
s.outBusInfo.numberOfOutputs = uint8(zeros(dimMethodName(1), 1)));
% update the outputsTypes
s.outBusInfo.outputsTypes = uint8(zeros(dimInputs(2)));
% update the outputsValues
s.outBusInfo.outputsValues = double(zeros(dimInputs(2)));

for i = 1:u1.numberOfMethods
    u1MethodName = u1.methodName(i,:);
    tempMethodName = uint8(zeros(1, dimMethodName(2)));
    u1NumberOfInputs = u1.numberOfInputs(i);
    u1InputsTypes = u1.inputsTypes(i,:);
Appendix B: Pipe connector source codes

Listing B2

```matlab
function [yL,yR] = extract(y0,uL,uR)

u1InputsValues = u1.inputsValues(i,:);
u1NumberOfOutputs = u1.outBusInfo.numberOfOutputs(i);
u1OutputsTypes = u1.outBusInfo.outputsTypes(i,:);
u1OutputsValues = u1.outBusInfo.outputsValues{i,:};
index1 = extractIndex(u1MethodName);
for j = 1:u2.numberOfMethods
    u2MethodName = u2.methodName(j,:);
    u2NumberOfInputs = u2.numberOfInputs(j);
    u2InputsTypes = u2.inputsTypes(j,:);
    u2NumberOfOutputs = u2.outBusInfo.numberOfOutputs(j);
    u2OutputsTypes = u2.outBusInfo.outputsTypes(j,:);
    u2OutputsValues = u2.outBusInfo.outputsValues{j,:};
    index2 = extractIndex(u2MethodName);
    if (u1NumberOfOutputs == u2NumberOfInputs)
        if (areEqual(u1OutputsTypes,u2InputsTypes) == 1)
            matchFound = true;
            %save the result in a the structure s
            s.isValidFlag = matchFound;
            % set the description flag to true
            s.descriptionFlag = true;
            % update the errorBusInfo
            s.errorBusInfo.errorCode = uint16(0);
            s.errorBusInfo.errorPath = tp;
            % update the number of methods
            s.numberOfMethods = k;
            % update the method name
            s.methodName(k,:) = u1MethodName;
            tempMethodName = composeName(s.methodName(k,:),u2MethodName);
            s.methodName(k,:) = tempMethodName;
            %update the number of inputs
            s.numberOfInputs(k) = u1NumberOfInputs;
            %update the inputsTypes
            s.inputsTypes(k,:) = u1InputsTypes;
            %update the inputsValues
            s.inputsValues(k,:) = u1InputsValues;
            %update the number of outputs
            s.outBusInfo.numberOfOutputs(k) = u2NumberOfOutputs;
            s.outBusInfo.outputsTypes(k,:) = u2OutputsTypes;
            s.outBusInfo.outputsValues(k,:) = u2OutputsValues;
            k = k + uint8(1);
        end
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%% UTILITIES
%This function is used to test if two vectors are equal
function vectEqual = areEqual(v1,v2)
vectEqual = all(v1==v2);

%This function is used to assemble two vector in one
function name = composeName(oldName, newName)
    name = uint8(zeros(size(oldName)));
    n = 1;
    a = 1;
    while (oldName(n) ~= uint8(0))
        name(n) = oldName(n);
        n = n+1;
    end
    if (oldName(n) == uint8(0))
        while (newName(a) ~= uint8(0))
            name(n) = newName(a);
            a = a + 1;
            n = n + 1;
        end
    end
end
```

Patrice Siatchoua – School of Computer Science - 2009
% This function is used to extract left and right structures to be passed
% to the left and right component given an input structure and description
% of the left and the right component

% Inputs: y0 [type{componentInbus},size{1}]
%         uL [type{componentOutBus},size{1}] description structure left
%         uR [type{componentOutBus},size{1}] description structure right
% Outputs: yL [type{componentInbus},size{1}] input for the left component
%          yR [type{componentInbus},size{1}] input for the right component

%Invariant1: Structure fields must be assigned in the same order
%on all control flow paths.

% (c) 2009, Patrice Siatchoua

dimMethodName = size(y0.methodName);
y0LastIndex = extractIndex(y0.methodName);
uLInd = 0;
uRInd = 0;

% @initialize %left
yL.descriptionFlag = false;
yL.numberOfMethods = uint8(0);
dimMethodNames = size(y0.methodName);
yL.methodName = uint8(zeros(dimMethodNames(1),dimMethodNames(2)));
yL.inBusInfo.numberOfInputs = uint8(0);
dimInputsTypes = size(y0.inBusInfo.inputsTypes);
yL.inBusInfo.inputsTypes = uint8(zeros(dimInputsTypes(1),dimInputsTypes(2)));
yL.inBusInfo.inputsValues = double(zeros(dimInputsTypes(1),dimInputsTypes(2)));

%right
yR = yL;

nextIndex1 = 0;
nextIndex2 = 0;
result1 = false;
result2 = false;

for i = 1:uL.numberOfMethods
  [nextIndex1,result1] = isPartOf(uL.methodName(i,:),y0.methodName,1);
  if (result1 == true)
    uLInd = double(i);
    for j = 1:uR.numberOfMethods
      uRLastIndex = extractIndex(uR.methodName(j,:));
      [nextIndex2,result2] = isPartOf(uR.methodName(j,:),y0.methodName,nextIndex1+1);
      if (result2 == true)
        uRInd = double(j);
        break;
      end
    end
  end
end
if (uLInd ~= 0 && uRInd ~= 0)
  break;
end

if (uLInd ~= 0 && uRInd ~= 0)
  %left
  yL.descriptionFlag = false;
  yL.numberOfMethods = uint8(1);
  yL.methodName = uL.methodName(uLInd,:);
  yL.inBusInfo.numberOfInputs = y0.inBusInfo.numberOfInputs;
  yL.inBusInfo.inputsTypes = y0.inBusInfo.inputsTypes;
  yL.inBusInfo.inputsValues = y0.inBusInfo.inputsValues;
  %right
  yR.descriptionFlag = false;
  yR.numberOfMethods = uint8(1);
  yR.methodName = uR.methodName(uRInd,:);
  yR.inBusInfo.numberOfInputs = uR.inBusInfo.numberOfInputs;
  yR.inBusInfo.inputsTypes = uR.inBusInfo.inputsTypes;
  yR.inBusInfo.inputsValues = uR.inBusInfo.inputsValues;
end

% This function check if vector u1 is part of vector u2 taking from position
% k, it returns the index nm corresponding to the last index where the two
%vectors match; it returns boolean bol true if the two vectors match to the
%extend of ul length
function [nm,bol] = isPartOf(u1,u2,k)
    nm = 0;
    bol = false;
    lgt = 0;
    for l = 1:length(u1)
        if (u1(l) == uint8(0))
            break;
        else
            lgt = lgt + 1;
        end
        if (u1(l) == u2(k+nm))
            nm = nm+1;
        end
    end
    if (nm == lgt)
        bol = true;
    end
end

Listing B3

function bol  = isAvailable(u,y0)
%#eml
% This function is used to check where there are methods in y0 that are in
% u
% Inputs: u [type{componentInbus},size{1}]
%         y0 [type{componentOutBus},size{1}]
% Outputs: bol [type{boolean},size{1}]
%%Invariant1: Structure fields must be assigned in the same order
%%on all control flow paths.
% (c) 2009, Patrice Siatchoua

dim = size(u.methodName);
dim1 = size(u.inBusInfo.inputsTypes);
k = uint8(1);
bol = false;
tpVar = 0;
aMethodName = uint8(zeros(1,dim(2)));
inputsTypes = uint8(zeros(1,dim1(2)));
inputsValues = zeros(1,dim1(2));
    for i = 1:u.numberOfMethods
        aMethodName = u.methodName(i,:);
        nberInputs = u.inBusInfo.numberOfInputs(i);
        inputsTypes = u.inBusInfo.inputsTypes(i,:);
        inputsValues = u.inBusInfo.inputsValues(i,:);
        for j = 1:y0.numberOfMethods
            if (all(aMethodName == y0.methodName(j,:)) == 1)
                if (nberInputs == y0.numberOfInputs(j))
                    if (all(inputsTypes == y0.inputsTypes(j,:)) == 1)
                        tpVar = 1;
                        break;
                    end
                end
            end
        end
    end
    if tpVar == 1
        bol = true;
    end
Appendix B: Pipe connector source codes

Listing B4

function y = copyResult(u,v)
%#eml
% This function is used to create a result y from 2 input structures u and v
% Inputs: u [type{componentOutbus},size{1}]  
%         v [type{componentInBus},size{1}]  
% Outputs: y [type{componentInBus},size{1}]  
% %Invariant1: Structure fields must be assigned in the same order on all control flow paths.
% %Invariant2: During computation, the number of methods in an output bus is always 1
% (c) 2009, Patrice Siatchoua

y.descriptionFlag = false;
y.numberOfMethods = uint8(0);
dimMethodName = size(v.methodName);
y.methodName = uint8(zeros(dimMethodName(1),dimMethodName(2)));
y.inBusInfo.numberOfInputs = uint8(0);
dimInputsTypes = size(v.inBusInfo.inputsTypes);
y.inBusInfo.inputsTypes = uint8(zeros(dimInputsTypes(1),dimInputsTypes(2)));
y.inBusInfo.inputsValues = double(zeros(dimInputsTypes(1),dimInputsTypes(2)));

y.descriptionFlag = false;
y.numberOfMethods = uint8(1);
y.methodName = v.methodName;
y.inBusInfo.numberOfInputs = u.outBusInfo.numberOfOutputs(1,:);
y.inBusInfo.inputsTypes = u.outBusInfo.outputsTypes(1,:);
y.inBusInfo.inputsValues = u.outBusInfo.outputsValues(1,:);

Listing B5

function s = updateResult(u1,u2)
%#eml
% This function is used to update the resulting structure after computation is performed in the attached components
% Inputs: u1 [type{componentInbus},size{1}]  
%         u2 [type{componentInBus},size{1}]  
% Outputs: s [type{componentOutBus},size{1}]  
% %Invariant1: Structure fields must be assigned in the same order on all control flow paths.
% (c) 2009, Patrice Siatchoua

dimMethodName = size(u2.methodName);
dimInputs = size(u2.inputsTypes);

% initialisation of the structure s
s.isValidFlag = true;
% set the description flag to true
s.descriptionFlag = false;
% update the errorBusInfo
s.errorBusInfo.errorCode = uint16(0);
theDim = size(u2.errorBusInfo.errorPath);
tp = EnumPathValues(zeros(theDim(1),1));
s.errorBusInfo.errorPath = tp;
% update the number of methods
s.numberOfMethods = uint8(0);
% update the method name
s.methodName = uint8(zeros(dimMethodName(1),1));
% update the number of inputs
s.numberOfInputs = uint8(zeros(dimMethodName(1),1));
% update the inputsTypes
s.inputsTypes = uint8(zeros(dimInputs(1),1));
% update the inputsValues
s.inputsValues = double(zeros(dimInputs));

% update the number of outputs
s.outBusInfo.numberOfOutputs = uint8(zeros(dimMethodName(1),1));
s.outBusInfo.outputsTypes = uint8(zeros(dimInputs));
s.outBusInfo.outputsValues = double(zeros(dimInputs));

s.isValidFlag = true;
s.descriptionFlag = false;
s.errorBusInfo.errorCode = u2.errorBusInfo.errorCode;
theDim = size(u2.errorBusInfo.errorPath);
  tp = EnumPathValues(zeros(theDim(1),1));
s.errorBusInfo.errorPath = tp;
s.numberOfMethods = u1.numberOfMethods;
s.methodName(1,:) = u1.methodName;
s.numberOfInputs(1) = u1.inBusInfo.numberOfInputs;
s.inputsTypes(1,:) = u1.inBusInfo.inputsTypes;
s.inputsValues(1,:) = u1.inBusInfo.inputsValues;
s.outBusInfo.numberOfOutputs(1) = u2.outBusInfo.numberOfOutputs(1);
s.outBusInfo.outputsTypes(1,:) = u2.outBusInfo.outputsTypes(1,:);
s.outBusInfo.outputsValues(1,:) = u2.outBusInfo.outputsValues(1,:);
Appendix C: Sequencer connector source codes

Listing C1

```matlab
function s = updateOutput(u1, u2)
% This function is used to assemble the description outputs of connected components into one single description structure
% Inputs: u1 [type{componentOutBus}, size{1}] component connected left
% u2 [type{componentOutBus}, size{1}] component connected right
% Output: s [type{componentOutBus}, size{1}]
% Invariant1: Structure fields must be assigned in the same order
% on all control flow paths.
% (c) 2009, Patrice Siatchoua

matchFound = false;
dimMethodName = size(u1.methodName);
u1MethodName = uint8(zeros(1, dimMethodName(2)));
dimInputs = size(u1.outBusInfo.outputsTypes); %same for inputs and outputs
u1InputsTypes = uint8(zeros(1, dimInputs(2)));
u1InputsValues = double(zeros(1, dimInputs(2)));
u1OutputsTypes = uint8(zeros(1, dimInputs(2)));
u1OutputsValues = double(zeros(1, dimInputs(2)));
u2MethodName = uint8(zeros(1, dimMethodName(2)));
u2InputsTypes = uint8(zeros(1, dimInputs(2)));
u2InputsValues = double(zeros(1, dimInputs(2)));
u2OutputsTypes = uint8(zeros(1, dimInputs(2)));
u2OutputsValues = double(zeros(1, dimInputs(2)));

tempMethodName = uint8(zeros(1, dimMethodName(2)));
tempInputsTypes = uint8(zeros(1, dimInputs(2)));
tempInputsValues = double(zeros(1, dimInputs(2)));
tempOutputsTypes = uint8(zeros(1, dimInputs(2)));
tempOutputsValues = double(zeros(1, dimInputs(2)));

k = uint8(1); % used for the number of methods in the resulting structure
%s.isValidFlag = matchFound;
%s.descriptionFlag = true;
% update the errorBusInfo
s.errorBusInfo.errorCode = uint16(0);
theDim = size(u1.errorBusInfo.errorPath);
.tp = EnumPathValues(zeros(theDim(1), 1));
s.errorBusInfo.errorPath = .tp;
% update the number of methods
s.numberOfMethods = uint8(0);
% update the method name
s.methodName = uint8(zeros(dimMethodName));
% update the number of inputs
s.numberOfInputs = uint8(zeros(dimMethodName(1),1));
% update the inputsTypes
s.inputsTypes = uint8(zeros(dimInputs));
% update the inputsValues
s.inputsValues = double(zeros(dimInputs));
% update the number of outputs
s.outBusInfo.numberOfOutputs = uint8(zeros(dimMethodName(1),1));
% update the outputsTypes
s.outBusInfo.outputsTypes = uint8(zeros(dimInputs));
% update the outputsValues
s.outBusInfo.outputsValues = double(zeros(dimInputs));
```

Patrice Siatchoua – School of Computer Science - 2009
for i = 1:u1.numberOfMethods
    u1MethodName = u1.methodName(i,:);
    u1NumberOfInputs = u1.numberOfInputs(i);
    u1InputsTypes = u1.inputsTypes(i,:);
    u1InputsValues = u1.inputsValues(i,:);
    u1NumberOfOutputs = u1.outBusInfo.numberOfOutputs(i);
    u1OutputsTypes = u1.outBusInfo.outputsTypes(i,:);
    u1OutputsValues = u1.outBusInfo.outputsValues{i,:};
for j = 1:u2.numberOfMethods
    u2MethodName = u2.methodName(j,:);
    u2NumberOfInputs = u2.numberOfInputs(j);
    u2InputsTypes = u2.inputsTypes(j,:);
    u2InputsValues = u2.inputsValues(j,:);
    u2NumberOfOutputs = u2.outBusInfo.numberOfOutputs(j);
    u2OutputsTypes = u2.outBusInfo.outputsTypes(j,:);
    u2OutputsValues = u2.outBusInfo.outputsValues{j,:};
% save the result in a the structure s
s.isValidFlag = true;
% set the description flag to true
s.descriptionFlag = true;
% update the errorBusInfo
s.errorBusInfo.errorCode = uint16(0);
s.errorBusInfo.errorPath = tp;
% update the number of methods
s.numberOfMethods = k;
% update the method name
s.methodName(k,:) = u1MethodName;
    tempMethodName = composeName(s.methodName(k,:),u2MethodName);
    s.methodName(k,:) = tempMethodName;
% update the number of inputs
s.numberOfInputs(k) = u1NumberOfInputs + u2NumberOfInputs;
% update the inputsTypes
s.inputsTypes(k,:) = u1InputsTypes;
    tempInputsTypes = composeName(s.inputsTypes(k,:),u2InputsTypes);
    s.inputsTypes(k,:) = tempInputsTypes;
% update the inputsValues
s.inputsValues(k,:) = u1InputsValues;
    tempInputsValues = double(composeNameDouble(s.inputsValues(k,:),u2InputsValues));
    s.inputsValues(k,:) = tempInputsValues;
% update the number of outputs
s.outBusInfo.numberOfOutputs(k) = u1NumberOfOutputs + u2NumberOfOutputs;
    s.outBusInfo.outputsTypes(k,:) = u1OutputsTypes;
    s.outBusInfo.outputsTypes(k,:) = tempOutputsTypes;
    s.outBusInfo.outputsValues(k,:) = u1OutputsValues;
    tempOutputsValues = double(composeNameDouble(s.outBusInfo.outputsValues(k,:),u2OutputsValues));
    s.outBusInfo.outputsValues(k,:) = tempOutputsValues;
    k = k + uint8(1);
end
end

%%%%%%%%%%%%%%%%%%%%%%%%% UTILITIES
% This function is used to assemble two vector in one
function name = composeName(oldName, newName)
    name = uint8(zeros(size(oldName)));
    n = 1;
    a = 1;
    while (oldName(n) ~= uint8(0))
        name(n) = oldName(n);
        n = n+1;
    end
    if (oldName(n) == uint8(0))
        while (newName(a) ~= uint8(0))
            name(n) = newName(a);
            a = a + 1;
            n = n + 1;
        end
    end
% This function is used to assemble two vector in one
% when the type is of input argument is double
function name = composeNameDouble(oldName, newName)
    name = double(zeros(size(oldName)));
n = 1;
a = 1;
while (oldName(n) ~= double(0))
    name(n) = oldName(n);
    n = n+1;
end
if (oldName(n) == double(0))
    while (newName(a) ~= double(0))
        name(n) = newName(a);
        a = a + 1;
        n = n + 1;
    end
end

Listing C2

function [yL,yR] = extract(y0,uL,uR)
%#eml
% This function is used to extract left and right structures to be passed
% to the left and right component given an input structure and description
% of the left and the right component
% Inputs: y0 [type{componentInbus},size{1}]
%         uL [type{componentOutBus},size{1}] description structure left
%         uR [type{componentOutBus},size{1}] description structure right
% Outputs: yL [type{componentInbus},size{1}] input for the left component
%          yR [type{componentInbus},size{1}] input for the right component
%\%Invariant1: Structure fields must be assigned in the same order
%\%on all control flow paths.
%\% (c) 2009, Patrice Siatchoua

dimMethodName = size(y0.methodName);
y0LastIndex = extractIndex(y0.methodName);
uLInd = 0;
uRInd = 0;
% %initialize
%left
yL.descriptionFlag = false;
yL.numberOfMethods = uint8(0);
dimMethodNames = size(y0.methodName);
yL.methodName = uint8(zeros(dimMethodNames(1),dimMethodNames(2)));
yL.inBusInfo.numberOfInputs = uint8(0);
dimInputsTypes = size(y0.inBusInfo.inputsTypes);
yL.inBusInfo.inputsTypes = uint8(zeros(dimInputsTypes(1),dimInputsTypes(2)));
yL.inBusInfo.inputsValues = double(zeros(dimInputsTypes(1),dimInputsTypes(2)));
%right
yR = yL;
nextIndex1 = 0;
nextIndex2 = 0;
result1 = false;
result2 = false;
for i = 1:uL.numberOfMethods
    [nextIndex1,result1] = isPartOf(uL.methodName(i,:),y0.methodName,1);
    if (result1 == true)
        uLInd = double(i);
        for j = 1:uR.numberOfMethods
            uRLastIndex = extractIndex(uR.methodName(j,:));
            [nextIndex2,result2] = isPartOf(uR.methodName(j,:),y0.methodName,nextIndex1+1);
            if (result2 == true)
                uRInd = double(j);
                break;
            end
        end
    end
end
if (uLInd ~= 0 && uRInd ~= 0)
    break;
end
if (uLInd ~= 0 && uRInd ~= 0)
%left
  yL.descriptionFlag = false;
yL.numberOfMethods = uint8(1);
yL.methodName = uL.methodName(uLInd,:);
yL.inBusInfo.numberOfInputs = uL.numberOfInputs(uLInd,:);
  for t=1:yL.inBusInfo.numberofInputs
    yL.inBusInfo.inputsValues(t) = y0.inBusInfo.inputsValues(t);
  end
%right
  yR.descriptionFlag = false;
yR.numberOfMethods = uint8(1);
yR.methodName = uR.methodName(uRInd,:);
yR.inBusInfo.numberOfInputs = uR.numberOfInputs(uRInd,:);
  for t=1:yR.inBusInfo.numberofInputs
    yR.inBusInfo.inputsValues(t) = y0.inBusInfo.inputsValues(t+yL.inBusInfo.numberOfInputs);
  end

%%%%%%%%%%%%%%%%%%%%%%%%% UTILITIES
function [nm,bol] = isPartOf(u1,u2,k)
  nm  = 0;
bol = false;
lgt = 0;
  for l = 1:length(u1)
    if (u1(l) == uint8(0))
      break;
    else
      lgt = lgt + 1;
      end
    if (u1(l) == u2(k+nm))
      nm = nm+1;
    end
  end
  if (nm == lgt)
    bol = true;
  end

Listing C3

Same as listing B3.

Listing C4

function s = updateResult(y,u1,u2)
%#eml
% This function is used to update the resulting structure after computation
% is performed in the attached components
% Inputs: y [type{componentInbus}, size{1}]  main data input
%         u1 [type{componentOutBus},size{1}] left component
%         u2 [type{componentOutBus},size{1}] right component
% Outputs: s [type{componentOutBus},size{1}] main data output
%#Invariant1: Structure fields must be assigned in the same order
%on all control flow paths.
% (c) 2009, Patrice Siatchoua

dimMethodName = size(u2.methodName);
dimInputs = size(u2.inputsTypes);
tempOutputsTypes = uint8(zeros(1,dimInputs(2)));
tempOutputsValues = double(zeros(1,dimInputs(2)));

% initialization of the structure s
s.isValidFlag = true;
% set the description flag to false
s.descriptionFlag = false;
% update the errorBusInfo
s.errorBusInfo.errorCode = uint16(0);
theDim = size(u2.errorBusInfo.errorPath);
 tp = EnumPathValues(zeros(theDim(1),1));
s.errorBusInfo.errorPath = tp;
% update the number of methods
s.numberOfMethods = uint8(0);
% update the inputsTypes
s.inputsTypes = uint8(zeros(dimInputs));
% update the inputsValues
s.inputsValues = double(zeros(dimInputs));
% update the number of outputs
s.outBusInfo.numberOfOutputs = uint8(zeros(dimMethodName(1),1));
% update the outputsTypes
s.outputsTypes = uint8(zeros(dimMethodName(1),1));
% update the outputsValues
s.outputsValues = double(zeros(dimMethodName(1),1));

% update the method name
s.methodName = uint8(zeros(dimMethodName));
% update the number of inputs
s.numberofInputs = uint8(zeros(dimMethodName(1),1));
% update the inputsTypes
s.inputsTypes = uint8(zeros(dimInputs));
% update the inputsValues
s.inputsValues = double(zeros(dimInputs));
% update the number of outputs
s.outBusInfo.numberOfOutputs = uint8(zeros(dimMethodName(1),1));
% update the outputsTypes
s.outputsTypes = uint8(zeros(dimMethodName(1),1));
% update the outputsValues
s.outputsValues = double(zeros(dimMethodName(1),1));

%%%%%%%%%%%%%%%%%%%%%%%%% UTILITIES
%This function is used to assemble two vector in one
function name = composeName(oldName, newName)

name = uint8(zeros(size(oldName)));
n = 1;
a = 1;
while (oldName(n) ~= uint8(0))
   name(n) = oldName(n);
   n = n+1;
end
if (oldName(n) == uint8(0))
   while (newName(a) ~= uint8(0))
      name(n) = newName(a);
      a = a + 1;
      n = n + 1;
   end
end
%This function is used to assemble two vector in one
%when the type is of input argument is double
function name = composeNameDouble(oldName, newName)

name = double(zeros(size(oldName)));
n = 1;
a = 1;
while (oldName(n) ~= double(0))
    name(n) = oldName(n);
    n = n+1;
end
if (oldName(n) == double(0))
    while (newName(a) ~= double(0))
        name(n) = newName(a);
        a = a + 1;
        n = n + 1;
    end
end
Appendix D: Selector connector source codes

Listing D1

```matlab
function s=updateOutput(u1,u2)
% This function is used to assemble the description outputs of connected
% components into one single description structure
% Inputs: u1 [type{componentOutBus},size{1}] component connected left
%         u2 [type{componentOutBus},size{1}] component connected right
% Output: s [type{componentOutBus},size{1}]
%%Invariant1: Structure fields must be assigned in the same order
% on all control flow paths.
% (c) 2009, Patrice Siatchoua

matchFound = false;
dimMethodName = size(u1.methodName);
u1MethodName = uint8(zeros(1,dimMethodName(2)));
u1Flag = ones(dimMethodName(1),1);
dimInputs = size(u1.outBusInfo.outputsTypes);  %same for inputs and outputs
u1InputsTypes = uint8(zeros(1,dimInputs(2)));
u1InputsValues = double(zeros(1,dimInputs(2)));
u1OutputsTypes = uint8(zeros(1,dimInputs(2)));
u1OutputsValues = double(zeros(1,dimInputs(2)));
dimMethodName = size(u2.methodName(1),1);
u2MethodName = uint8(zeros(1,dimMethodName(2)));
u2Flag = ones(dimMethodName(1),1);
u2InputsTypes = uint8(zeros(1,dimInputs(2)));
u2InputsValues = double(zeros(1,dimInputs(2)));
u2OutputsTypes = uint8(zeros(1,dimInputs(2)));
u2OutputsValues = double(zeros(1,dimInputs(2)));

tempMethodName = uint8(zeros(1,dimMethodName(2)));
tempInputsTypes = uint8(zeros(1,dimInputs(2)));
tempInputsValues = double(zeros(1,dimInputs(2)));
tempOutputsTypes = uint8(zeros(1,dimInputs(2)));
tempOutputsValues = double(zeros(1,dimInputs(2)));

k = uint8(1); % used for the number of methods in the resulting structure

% initialisation of the structure s
s.isValidFlag = matchFound;
% set the description flag to true
s.descriptionFlag = true;
% update the errorBusInfo
s.errorBusInfo.errorCode = uint16(0);
theDim = size(u1.errorBusInfo.errorPath);
 tp = EnumPathValues(zeros(theDim(1),1));
s.errorBusInfo.errorPath = tp;
% update the number of methods
s.numberOfMethods = uint8(0);
% update the method name
s.methodName = uint8(zeros(dimMethodName(1),1));
% update the number of inputs
s.numberOfInputs = uint8(zeros(dimMethodName(1),1));
% update the inputsTypes
s.inputsTypes = uint8(zeros(dimInputs(1),1));
% update the inputsValues
s.inputsValues = double(zeros(dimInputs(1),1));
% update the number of outputs
s.numberOfOutputs = uint8(zeros(dimMethodName(1),1));
% update the outputsTypes
s.outBusInfo.outputsTypes = uint8(zeros(dimInputs(1),1));
% update the outputsValues
s.outBusInfo.outputsValues = double(zeros(dimInputs(1),1));
```

Appendix D: Selector connector source codes

Listing D1
for i = 1:u1.numberOfMethods
    u1MethodName = u1.methodName(i,:);
    u1NumberOfInputs = u1.numberOfInputs(i);
    u1InputsTypes = u1.inputsTypes(i,:);
    u1InputsValues = u1.inputsValues(i,:);
    u1NumberOfOutputs = u1.outBusInfo.numberOfOutputs(i);
    u1OutputsTypes = u1.outBusInfo.outputsTypes(i,:);
    u1OutputsValues = u1.outBusInfo.outputsValues{i,:};
    methodFound = false;
    for j = 1:u2.numberOfMethods
        u2MethodName = u2.methodName(j,:);
        u2NumberOfInputs = u2.numberOfInputs(j);
        u2InputsTypes = u2.inputsTypes(j,:);
        u2InputsValues = u2.inputsValues(j,:);
        u2NumberOfOutputs = u2.outBusInfo.numberOfOutputs(j);
        u2OutputsTypes = u2.outBusInfo.outputsTypes(j,:);
        u2OutputsValues = u2.outBusInfo.outputsValues{j,:};
        s.isValidFlag = true;
        s.descriptionFlag = true;
        s.errorBusInfo.errorCode = uint16(0);
        s.errorBusInfo.errorPath = tp;
        s.numberOfMethods = k;
        if ((areEqual(u1MethodName,u2MethodName) == 1) && ...
            (areEqual(u1NumberOfInputs,u2NumberOfInputs) == 1) && ...
            (areEqual(u1InputsTypes,u2InputsTypes) == 1) && ...
            (areEqual(u1NumberOfOutputs,u2NumberOfOutputs) == 1) && ...
            (areEqual(u1OutputsTypes,u2OutputsTypes) == 1))
            s.methodName(k,:) = u1MethodName;
            s.numberOfInputs(k) = u1NumberOfInputs + 1;
            s.inputsTypes(k,:) = u1.inputsTypes(i,:);
            s.inputsTypes(k,u1NumberOfInputs+1) = uint8(9); % double
            s.inputsValues(k,:) = u1.inputsValues(i,:);
            s.outBusInfo.numberOfOutputs(k) = u1NumberOfOutputs;
            s.outBusInfo.outputsTypes(k,:) = u1.outBusInfo.outputsTypes(i,:);
            s.outBusInfo.outputsValues(k,:) = u1.outBusInfo.outputsValues{i,:};
            u2Flag(j) = 0;
            k = k + uint8(1);
            methodFound = true;
            break;
        end
    end
    if (methodFound == false)
        s.methodName(k,:) = u1MethodName;
        s.numberOfInputs(k) = u1NumberOfInputs;
        s.inputsTypes(k,:) = u1.inputsTypes(i,:);
        s.inputsValues(k,:) = u1.inputsValues(i,:);
        s.outBusInfo.numberOfOutputs(k) = u1NumberOfOutputs;
        s.outBusInfo.outputsTypes(k,:) = u1.outBusInfo.outputsTypes(i,:);
        s.outBusInfo.outputsValues(k,:) = u1.outBusInfo.outputsValues{i,:};
        k = k + uint8(1);
    end
end
for j = 1:u2.numberOfMethods
    if (u2Flag(j) == 0)
        s.isIsValidFlag = true;
        s.descriptionFlag = true;
        s.errorBusInfo.errorCode = uint16(0);
        s.errorBusInfo.errorPath = tp;
        s.numberOfMethods = k;
        s.methodName(k,:) = u2.methodName(j,:);
        s.numberOfInputs(k) = u2.numberOfInputs(j);
        s.inputsTypes(k,:) = u2.inputsTypes(j,:);
        s.inputsValues(k,:) = u2.inputsValues(j,:);
        s.outBusInfo.numberOfOutputs(k) = u2.outBusInfo.numberOfOutputs(j);
        s.outBusInfo.outputsTypes(k,:) = u2.outBusInfo.outputsTypes(j,:);
        s.outBusInfo.outputsValues(k,:) = u2.outBusInfo.outputsValues{j,:};
    end
end

% %%%%%%%%%%%%%%%%%%%%%%%%% UTILITIES
% This function check that the two vectors are equal
function vectEqual = areEqual(v1,v2)
    vectEqual = all(v1==v2);
Listing D2

function [y,dir] = extract(y0,uL,uR,a,c,d)
%%eml
% This function is used to extract left or right structures to be passed
% to the left or right component given an input structure and description
% of the left and the right component as well as some expression
% representing user choice
% Inputs: y0 [type{componentInbus},size{1}]
%         uL [type{componentOutBus},size{1}] description structure left
%         uR [type{componentOutBus},size{1}] description structure right
% a  [type{uint8},size{1}] is component 2 checked?
% b  [type{uint8},size{1}] which condition operator was choosen?
% c  [type{double},size{1}] condition value
% Outputs: y [type{componentInBus},size{1}] input for choosen component
%          dir [type{uint8},size{1}] left component or right component
%Invariant1: Structure fields must be assigned in the same order
%on all control flow paths.
% (c) 2009, Patrice Siatchoua

foundInUl = false;
foundInUr = false;
res = 1;
dir = uint8(1);
y = y0;
for i = 1:uL.numberOfMethods
    if ((areEqual(y0.methodName(1,:),uL.methodName(i,:)) == 1) && ...
        (y0.inBusInfo.numberOfInputs == uL.inBusInfo.numberOfInputs(i)))
        y = y0;
        dir = uint8(1);
        return;
    elseif ((areEqual(y0.methodName(1,:),uL.methodName(i,:)) == 1) && ...
        (y0.inBusInfo.numberOfInputs == uL.inBusInfo.numberOfInputs(i)+1))
        foundInUl = true;
        break;
    end
end
if foundInUl == true
    for i = 1:uR.numberOfMethods
        if ((areEqual(y0.methodName(1,:),uR.methodName(i,:)) == 1) && ...
            (y0.inBusInfo.numberOfInputs == uR.inBusInfo.numberOfInputs(i)))
            y = y0;
            dir = uint8(2);
            return;
        elseif ((areEqual(y0.methodName(1,:),uR.methodName(i,:)) == 1) && ...
            (y0.inBusInfo.numberOfInputs == uR.inBusInfo.numberOfInputs(i)+1))
            foundInUr = true;
            break;
        end
    end
    else
    y = y0;
    dir = uint8(2);
end
if (foundInUl == true && foundInUr == true)
dir = uint8(evalExpr(a,y0.inBusInfo.inputsValues(1,y0.inBusInfo.numberOfInputs),c,d));
y.descriptionFlag = y0.descriptionFlag;
y.numberOfMethods = y0.numberOfMethods;
y.methodName = y0.methodName;
y.inBusInfo.numberOfInputs = y0.inBusInfo.numberOfInputs - uint8(1);
y.inBusInfo.inputTypes = y0.inBusInfo.inputTypes;
y.inBusInfo.inputsTypes(y0.inBusInfo.numberOfInputs) = uint8(0);
y.inBusInfo.inputsValues = y0.inBusInfo.inputsValues;
y.inBusInfo.inputsValues(y0.inBusInfo.numberOfInputs) = uint8(0);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% UTILITIES
\begin{verbatim}
%This function check if the two vectors are equal
function vectEqual = areEqual(v1,v2)
    vectEqual = all(v1==v2);

%This function evaluate the expression condVar-condOp-condVal
%and return 1 if the condition is false or 2 otherwise
%This is helpful to decide with component to select
function res = evalExpr(comp1,condVar,condOp,condVal)

    bolVal = false;
    res = 1; % 1 means first component
    % 2 means second component
    %is Expression (condVar condOp condVal) true?
    if condOp == uint8(1)
        bolVal = (condVar == condVal);
    elseif condOp == uint8(2)
        bolVal = (condVar <= condVal);
    elseif condOp == uint8(3)
        bolVal = (condVar >= condVal);
    elseif condOp == uint8(4)
        bolVal = (condVar < condVal);
    elseif condOp == uint8(5)
        bolVal = (condVar > condVal);
    elseif condOp == uint8(6)
        bolVal = (condVar ~= condVal);
    end

    if (comp1 == uint8(1)) % 'on'
        if bolVal == true
            res = 2;
        else
            res = 1;
        end
    else
        if bolVal == true
            res = 1;
        else
            res = 2;
        end
    end
\end{verbatim}

**Listing D3**

Same as listing B3.
Listing E1

```matlab
function s=updateOutput(u1,varLoop)
% This function is used to assemble the description outputs of connected components into one single description structure
% Inputs: u1 [type{componentOutBus},size{1}] component connected left
% varLoop [type{uint8},size{1}] represents the number of iterations
% Output: s [type{componentOutBus},size{1}]
%%Invariant1: Structure fields must be assigned in the same order on all control flow paths.
% (c) 2009, Patrice Siatchoua

dimMethodName = size(u1.methodName);
dimInputs = size(u1.outBusInfo.outputsTypes);
isValid = false;
numberOfMethods = uint8(0);
tab = (zeros(size(u1.outBusInfo.numberOfOutputs)));
for n = 1:u1.numberOfMethods
    if (u1.outBusInfo.numberOfOutputs(n)* varLoop <= dimInputs(2))
        tab(numberOfMethods+1) = n;
        numberOfMethods = numberOfMethods + 1;
    end
end
s = u1;
if ((varLoop < 1) || numberOfMethods == 0)
    % Initialization of the structure s
    s.isValidFlag = false;
    % set the description flag to true
    s.descriptionFlag = true;
    % update the errorBusInfo
    s.errorBusInfo.errorCode = uint16(1);
    theDim = size(u1.errorBusInfo.errorPath);
    tp = EnumPathValues(zeros(theDim(1),1));
    s.errorBusInfo.errorPath = tp;
    % update the number of methods
    s.numberOfMethods = uint8(0);
    % update the method name
    s.methodName = uint8(zeros(dimMethodName));
    %update the number of inputs
    s.numberOfInputs = uint8(zeros(dimMethodName(1),1));
    %update the inputsTypes
    s.inputsTypes = uint8(zeros(dimInputs));
    %update the inputsValues
    s.inputsValues = double(zeros(dimInputs));
    %update the number of outputs
    s.outBusInfo.numberOfOutputs = uint8(zeros(dimMethodName(1),1));
    s.outBusInfo.outputsTypes = uint8(zeros(dimInputs));
    s.outBusInfo.outputsValues = double(zeros(dimInputs));
else
    s.isValidFlag = true;
    % set the description flag to true
    s.descriptionFlag = true;
    % update the errorBusInfo
    s.errorBusInfo.errorCode = uint16(0);
    theDim = size(u1.errorBusInfo.errorPath);
    tp = EnumPathValues(zeros(theDim(1),1));
    s.errorBusInfo.errorPath = tp;
    % update the number of methods
    s.numberOfMethods = numberOfMethods;
end
```
for i = 1:s.numberOfMethods
  % update the method name
  s.methodName(i,:) = u1.methodName(tab(i),:);
  % update the number of inputs
  s.numberOfInputs(i) = u1.numberOfInputs(tab(i));
  % update the inputsTypes
  s.inputsTypes(i,:) = u1.inputsTypes(tab(i),:);
  % update the inputsValues
  s.inputsValues(i,:) = u1.inputsValues(tab(i),:);
  % update the number of outputs
  s.outBusInfo.numberOfOutputs(i) = u1.outBusInfo.numberOfOutputs(tab(i)) *
    varLoop;
  m = 1;
  sOutputsValues = double(zeros(1,dimInputs(2)));
  sOutputsTypes = uint8(zeros(1,dimInputs(2)));
  for j = 1:varLoop
    for k = 1:u1.outBusInfo.numberOfOutputs(tab(i))
      sOutputsTypes(m) = u1.outBusInfo.outputsTypes(tab(i),k);
      sOutputsValues(m) = u1.outBusInfo.outputsValues(tab(i),k);
      m = m + 1;
    end
  end
  s.outBusInfo.outputsTypes(i,:) = sOutputsTypes;
  s.outBusInfo.outputsValues(i,:) = sOutputsValues;
  m = 1;
end

Listing E2

Same as Listing B3

Listing E3

function s = updateResult(u1,a)
  % This function is used to update the resulting structure after computation
  % is performed in the attached components
  % Inputs: u1 [type{componentOutbus},size{1}] left component
  %         a [type{uint8},size{1}] counter iteration
  % Outputs: s [type{componentOutBus},size{1}] main data output
  % Invariant1: Structure fields must be assigned in the same order
  % on all control flow paths.
  % (c) 2009, Patrice Siatchoua
  persistent outS;
  persistent m;
  dimMethodName = size(u1.methodName);
  dimInputs = size(u1.outBusInfo.outputsTypes);
  varLoop = a;
  if isempty(outS) || (varLoop == 1)
    outS = u1;
    m = 1;
  end
  outS.isValidFlag = true;
  % set the description flag to true
  outS.descriptionFlag = false;
  % update the errorBusInfo
  outS.errorBusInfo.errorCode = uint16(0);
  theDim = size(u1.errorBusInfo.errorPath);
  tp = EnumPathValues(zeros(theDim(1),1));
  outS.errorBusInfo.errorPath = tp;
  % update the number of methods
  outS.numberOfMethods = u1.numberOfMethods;
  % update the method name
  outS.methodName = u1.methodName;
Appendix E: Loop source codes

% update the number of inputs
outS.numberOfInputs = u1.numberOfInputs;
% update the inputsTypes
outS.inputsTypes = u1.inputsTypes;
% update the inputsValues
outS.inputsValues = u1.inputsValues;
% update the number of outputs
outS.outBusInfo.numberOfOutputs = u1.outBusInfo.numberOfOutputs * varLoop;
   for i = 1:outS.numberOfMethods
      for k = 1:u1.outBusInfo.numberOfOutputs(i)
         outS.outBusInfo.outputsTypes(i,m) = u1.outBusInfo.outputsTypes(i,k);
         outS.outBusInfo.outputsValues(i,m) = u1.outBusInfo.outputsValues(i,k);
         m = m + 1;
      end
   end
s = outS;
Appendix F: Input and output structure definition

```matlab
function newVersionComponentBuses()
% NEWVERSIONCOMPONENTBUSES initializes a set of bus objects in the MATLAB base workspace
% (c) Patrice Siatchoua, 2009
% Bus object: componentInBus
clear elems;
elems(1) = Simulink.BusElement;
elems(1).Name = 'descriptionFlag';
elems(1).Dimensions = 1;
elems(1).DataType = 'boolean';
elems(1).SampleTime = -1;
elems(1).Complexity = 'real';
elems(1).SamplingMode = 'Sample based';
elems(2) = Simulink.BusElement;
elems(2).Name = 'numberOfMethods';
elems(2).Dimensions = 1;
elems(2).DataType = 'uint8';
elems(2).SampleTime = -1;
elems(2).Complexity = 'real';
elems(2).SamplingMode = 'Sample based';
elems(3) = Simulink.BusElement;
elems(3).Name = 'methodName';
elems(3).Dimensions = [1 4000];
elems(3).DataType = 'uint8';
elems(3).SampleTime = -1;
elems(3).Complexity = 'real';
elems(3).SamplingMode = 'Sample based';
elems(4) = Simulink.BusElement;
elems(4).Name = 'inBusInfo';
elems(4).Dimensions = 1;
elems(4).DataType = 'subBusInput';
elems(4).SampleTime = -1;
elems(4).Complexity = 'real';
elems(4).SamplingMode = 'Sample based';
componentInBus = Simulink.Bus;
componentInBus.Description = sprintf('The componentInBus is used to carry information to
a component irrespective of the kind of component (either atomic or composite), the
whole description of the component is given in the file componentBuses.mat');
componentInBus.Elements = elems;
assignin('base', 'componentInBus', componentInBus)
% Bus object: componentOutBus
clear elems;
elems(1) = Simulink.BusElement;
elems(1).Name = 'isValidFlag';
elems(1).Dimensions = 1;
elems(1).DataType = 'boolean';
elems(1).SampleTime = -1;
elems(1).Complexity = 'real';
elems(1).SamplingMode = 'Sample based';
elems(2) = Simulink.BusElement;
elems(2).Name = 'descriptionFlag';
elems(2).Dimensions = 1;
elems(2).DataType = 'boolean';
elems(2).SampleTime = -1;
elems(2).Complexity = 'real';
elems(2).SamplingMode = 'Sample based';
```

Patrice Siatchoua – School of Computer Science - 2009 80
Appendix F: Input and output structure definition

elems(2).Complexity = 'real';
elems(2).SamplingMode = 'Sample based';

elems(3) = Simulink.BusElement;
elems(3).Name = 'errorBusInfo';
elems(3).Dimensions = 1;
elems(3).DataType = 'subBusError';
elems(3).SampleTime = -1;
elems(3).Complexity = 'real';
elems(3).SamplingMode = 'Sample based';

elems(4) = Simulink.BusElement;
elems(4).Name = 'numberOfMethods';
elems(4).Dimensions = 1;
elems(4).DataType = 'uint8';
elems(4).SampleTime = -1;
elems(4).Complexity = 'real';
elems(4).SamplingMode = 'Sample based';

elems(5) = Simulink.BusElement;
elems(5).Name = 'methodName';
elems(5).Dimensions = [100 4000];
elems(5).DataType = 'uint8';
elems(5).SampleTime = -1;
elems(5).Complexity = 'real';
elems(5).SamplingMode = 'Sample based';

elems(6) = Simulink.BusElement;
elems(6).Name = 'numberOfInputs';
elems(6).Dimensions = [100 1];
elems(6).DataType = 'uint8';
elems(6).SampleTime = -1;
elems(6).Complexity = 'real';
elems(6).SamplingMode = 'Sample based';

elems(7) = Simulink.BusElement;
elems(7).Name = 'inputsTypes';
elems(7).Dimensions = [100 100];
elems(7).DataType = 'uint8';
elems(7).SampleTime = -1;
elems(7).Complexity = 'real';
elems(7).SamplingMode = 'Sample based';

elems(8) = Simulink.BusElement;
elems(8).Name = 'inputsValues';
elems(8).Dimensions = [100 100];
elems(8).DataType = 'double';
elems(8).SampleTime = -1;
elems(8).Complexity = 'real';
elems(8).SamplingMode = 'Sample based';

elems(9) = Simulink.BusElement;
elems(9).Name = 'outBusInfo';
elems(9).Dimensions = 1;
elems(9).DataType = 'subBusOutput';
elems(9).SampleTime = -1;
elems(9).Complexity = 'real';
elems(9).SamplingMode = 'Sample based';

componentOutBus = Simulink.Bus;
componentOutBus.HeaderFile = '';
componentOutBus.Description = sprintf('The componentOutBus is used to carry information
to a component irrespective of the kind of component (either atomic or composite), the
whole description of the component is given in the file componentBuses.mat');
componentOutBus.Elements = elems;
assignin('base', 'componentOutBus', componentOutBus)

% Bus object: subBusError
clear elems;
elems(1) = Simulink.BusElement;
elems(1).Name = 'errorCode';
elems(1).Dimensions = 1;
elems(1).DataType = 'uint16';
elems(1).SampleTime = -1;
elems(1).Complexity = 'real';
elems(1).SamplingMode = 'Sample based';
Appendix F: Input and output structure definition

elems(2) = Simulink.BusElement;
elems(2).Name = 'errorPath';
elems(2).Dimensions = [100 1];
elems(2).DataType = 'Enum: EnumPathValues';
elems(2).SampleTime = -1;
elems(2).Complexity = 'real';
elems(2).SamplingMode = 'Sample based';
subBusError = Simulink.Bus;
subBusError.HeaderFile = '';
subBusError.Description = sprintf('');
subBusError.Elements = elems;
assignin('base', 'subBusError', subBusError)

% Bus object: subBusInput
clear elems;
elems(1) = Simulink.BusElement;
elems(1).Name = 'numberofInputs';
elems(1).Dimensions = 1;
elems(1).DataType = 'uint8';
elems(1).SampleTime = -1;
elems(1).Complexity = 'real';
elems(1).SamplingMode = 'Sample based';

subBusInput = Simulink.Bus;
subBusInput.HeaderFile = '';
subBusInput.Description = sprintf('');
subBusInput.Elements = elems;
assignin('base', 'subBusInput', subBusInput)

% Bus object: subBusOutput
clear elems;
elems(1) = Simulink.BusElement;
elems(1).Name = 'numberofOutputs';
elems(1).Dimensions = [100 1];
elems(1).DataType = 'uint8';
elems(1).SampleTime = -1;
elems(1).Complexity = 'real';
elems(1).SamplingMode = 'Sample based';

subBusOutput = Simulink.Bus;
subBusOutput.HeaderFile = '';
subBusOutput.Description = sprintf('');
subBusOutput.Elements = elems;
assignin('base', 'subBusOutput', subBusOutput)
Appendix G: MATLAB processing file

% Script used to process the result of the simulation
% This script will create a human readable file name
% © Patrice Siatchoua, 2009

filename = ['testResult' datestr(now, 'yyyymmddTHHMMSS')];
matFile = [filename, '.mat'];

save filename outIsValidFlag outDescriptionFlag outErrorCode outErrorPath...
outNumberOfMethods outMethodNames outNumberOfInputs outInputsTypes...
outInputsValues outNumberOfOutputs outOutputsTypes outOutputsValues;

copyfile('filename.mat', matFile);
delete('filename.mat');

textFile = [filename, '.txt'];
fid = fopen(textFile, 'wt');
fprintf(fid, '{ValidFlag: %u\n', outIsValidFlag.signals.values);
if (outIsValidFlag.signals.values == 0)
    fprintf(fid, 'ValidFlag: False\n');
else
    fprintf(fid, 'ValidFlag: True\n');
end
fprintf(fid, '{errorCode: %u\n', outErrorCode.signals.values);
fprintf(fid, '{errorPath:\n';
i = 1;
if (outErrorPath.signals.values(1) == 0)
    fprintf(fid, '{\%\%s\n', getErrorPosition(outErrorPath.signals.values(1)));
else
    while (outErrorPath.signals.values(i) ~= 0)
        fprintf(fid, '{\%\%s\n', getErrorPosition(outErrorPath.signals.values(i)));
        i = i+1;
    end
end
fprintf(fid, '{\n');
if (outDescriptionFlag.signals.values == 0)
    fprintf(fid, '{descriptionFlag: False\n');
else
    fprintf(fid, '{descriptionFlag: True\n');
end
fprintf(fid, '{\n');
fprintf(fid, '{numberOf methods: %u\n', outNumberOfMethods.signals.values);
fprintf(fid, '{\n');
fprintf(fid, '{method name(s):\n';
for i=1:double(outNumberOfMethods.signals.values)
    fprintf(fid, '{\%\%s\n', deblank(char(outMethodNames.signals.values(i,:))));
end
fprintf(fid, '{\n');
fprintf(fid, '{numberOf inputs for each method:\n';
for i=1:double(outNumberOfMethods.signals.values)
    fprintf(fid, '{\%\%s\n', outNumberOfInputs.signals.values(i));
end
fprintf(fid, '{\n');
fprintf(fid, '{inputs types: inputs values}\n');
for i=1:double(outNumberOfMethods.signals.values)
    if (outNumberOfInputs.signals.values(i) == 0)
        str = getTypeValueStr(outInputsTypes.signals.values(i, 1),...
Appendix G: MATLAB processing file

outInputsValues.signals.values(i,1));
fprintf(fid,'\t%s',str);
else
    for j=1:outNumberOfInputs.signals.values(i)
        str = getTypenValueStr(outInputsTypes.signals.values(i,j),...
            outInputsValues.signals.values(i,j));
        fprintf(fid,'\t%s',str);
    end
end
fprintf(fid,'\n');
end
fprintf(fid,'\n');
fprintf(fid, 'number of outputs for each method:\n');
for i=1:double(outNumberOfMethods.signals.values)
    fprintf(fid, '\t\u', outNumberOfOutputs.signals.values(i));
end
fprintf(fid,'\n');
fprintf(fid, '(outputs types: outputs values)\n');
for i=1:double(outNumberOfMethods.signals.values)
    if (outNumberOfOutputs.signals.values(i) == 0)
        str = getTypenValueStr(outOutputsTypes.signals.values(i,1),...
            outOutputsValues.signals.values(i,1));
        fprintf(fid,'\t%s',str);
    else
        for j=1:double(outNumberOfOutputs.signals.values(i))
            str = getTypenValueStr(outOutputsTypes.signals.values(i,j),...
                outOutputsValues.signals.values(i,j));
            fprintf(fid,'\t%s',str);
        end
    end
end
fclose(fid);
movefile(matFile,'ResultsDirectory');
movefile(textFile,'ResultsDirectory');
References