Francesco Stefanni

A Design & Verification Methodology for Networked Embedded Systems

Ph.D. Thesis

April 7, 2011

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
Prof. Franco Fummi

Co-Advisor:
Assistant Professor Davide Quaglia

Series N°: TD-04-11

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

Γνῶθι σεαυτόν,
ἕν οίδα ὅτι ούδὲν οίδα.

Abstract

Nowadays, Networked Embedded Systems (NES's) are a pervasive technology. Their use ranges from communication, to home automation, to safety critical fields. Their increasing complexity requires new methodologies for efficient design and verification phases. This work presents a generic design flow for NES's, supported by the implementation of tools for its application. The design flow exploits the SystemC language, and considers the network as a design space dimension. Some extensions to the base methodology have been performed to consider the presence of a middleware as well as dependability requirements. Translation tools have been implemented to allow the adoption of the proposed methodology with designs written in other HDL's.

Contents

1 Introduction ... 1
1.1 Thesis structure ... 5

2 Background ... 7
2.1 Networked Embedded Systems ... 7
2.1.1 Communication protocols ... 8
2.2 System Level Design ... 9
2.2.1 Top-down design flow ... 10
2.2.2 Hardware Description Languages ... 11
2.2.3 Abstract Middleware Environment ... 12
2.3 Assessing embedded systems correctness ... 13
2.3.1 Fault Injection and Fault Simulation ... 14

3 Proposed design & verification methodology ... 15
3.1 Methodology overview ... 15
3.2 Example of design flow adoption ... 18

4 Communication Aware Specification and Synthesis Environment ... 21
4.1 Related Work ... 22
4.2 Design flow for network synthesis ... 23
4.2.1 High-level system specification languages ... 24
4.3 The formal network model ... 25
4.3.1 Tasks ... 27
4.3.2 Data Flows ... 27
4.3.3 Nodes ... 28
4.3.4 Abstract Channels ... 28
4.3.5 Zones ... 29
4.3.6 Contiguities ... 30
4.3.7 Graph representation ... 31
4.3.8 Relationships between entities ... 31
4.4 Network synthesis ... 32
4.4.1 Problem formulation ... 32
4.4.2 Network synthesis methodology ... 33

4.4.3 Network synthesis taxonomy 33
4.5 Case study I: Temperature control 34
4.5.1 Tasks, data flows and zones 34
4.5.2 Task assignment 37
4.5.3 Node assignment 37
4.5.4 Assignment of abstract channels 38
4.6 Case study II: Matrix multiplication 39
4.6.1 Network specification 40
4.6.2 Network synthesis 40
4.6.3 Result of the synthesis process 41
4.7 Conclusions and future work 42

5 Integration of a Middleware in the CASSE methodology 43
5.1 Proposed Methodology 44
5.1.1 Extraction of Requirements from the Application 46
5.2 Building the Middleware Library 47
5.2.1 Modeling Middleware Communication Schema 47
5.2.2 MW Computation & Communication Requirements 48
5.3 Conclusions 49

6 Dependability modeling into CASSE 51
6.1 Proposed Methodology 52
6.1.1 Modeling of Dependability Issues 53
6.1.2 Dependability Constraints 54
6.1.3 Network Synthesis & Dependability Analysis 55
6.2 Case Study 55
6.3 Conclusions 58

7 Network Fault Model 59
7.1 Background 61
7.1.1 Fault modeling and simulation 61
7.1.2 Representation of network interactions 61
7.1.3 Single State Transition Fault Model 63
7.2 Definition of the Network Fault Model 64
7.2.1 FSA of the Abstract Channel 64
7.2.2 Fault injection policy 67
7.2.3 Test with standard protocols 67
7.2.4 The Time-varying Network Fault Model 69
7.2.5 TNFM fault activation policy 70
7.2.6 Observability 70
7.3 The simulation platform 71
7.3.1 Network Fault Simulation Library 72
7.4 Case studies 72
7.4.1 Temperature monitoring 72
7.4.2 Networked control system 74
7.4.3 Testbench qualification for a networked control system 76
7.5 Conclusions 76

8 Propagation Analysis Engine 79
8.1 Framework 80
8.2 Open issues 80
8.2.1 The functional fault model 81
8.2.2 Functional fault parallelization 82
8.2.3 The parallel simulation engine 83
8.2.4 The simulation kernel and the simulation language 84
8.3 Optimizations 85
8.3.1 Optimized inputs management 86
8.3.2 Mux computation optimization 86
8.3.3 Splitting the netlist logic cones 86
8.3.4 Optimizing the flops computations 86
8.3.5 Dealing with the compiler 87
8.3.6 The four value logic 87
8.3.7 Function inlining 87
8.4 Experimental results 88
8.5 Concluding remarks 89

9 HIF Suite 91
9.1 Related Work 92
9.2 HIFSuite Overview 93
9.3 HIF Core-Language and APIs 95
9.3.1 HIF Basic Elements 95
9.3.2 System Description by using HIF 96
9.3.3 HIF Application Programming Interfaces 98
9.3.4 HIF Semantics 100
9.4 Conversion Tools 102
9.4.1 The front-end and back-end conversion tools 102
9.4.2 HDL types 103
9.4.3 HDL cast and type conversion functions 104
9.4.4 HDL operators 105
9.4.5 HDL structural statements 106
9.4.6 HDL declaration semantics 106
9.5 Concluding Remarks 108

10 SystemC Network Simulation Library 109
10.1 Related work 111
10.2 SCNSL architecture 113
10.2.1 Main components 114
10.3 Issues in implementing SCNSL 116
10.3.1 Tasks and nodes 116
10.3.2 Transmission validity assessment 117
10.3.3 Simulation of RTL models 117
10.3.4 The notion of packet 118
10.3.5 RTL & TLM models coexistence 118
10.3.6 Simulation planning 118
10.3.7 Implementation of network protocols and channels 119

10.4 Experimental results . . . 120
10.5 Conclusions . . . 121

11 A final case study . . . 123
11.1 Case study description . . . 123
11.2 System View modeling . . . 123
11.3 Choosing the middleware . . . 124
11.4 CASSE modeling and network synthesis . . . 125
11.5 Remaining design steps . . . 126
11.6 Methodology validation . . . 126
11.7 Conclusions . . . 126

12 Conclusions . . . 127

A Publications . . . 129
A.1 Articles published on journals . . . 129
A.2 Papers published in conference proceedings . . . 129
A.3 Other publications . . . 130

B Acknowledgments . . . 131

C Sommario (in Italian) . . . 133

References . . . 135

1 Introduction

Today's distributed applications are becoming more and more complex, involving many different tasks spread over hundreds or thousands of networked embedded systems connected through different types of channels and protocols [58]. In this context, computer-aided design should be fruitfully applied not only to each node, as currently done in the context of electronic systems design, but also to the communication infrastructure among them.

For instance, the scenario depicted in Figure 1.1, is related to the temperature control application of a skyscraper. In each room, there is at least one sensor (in Figure denoted by *S*) which keeps track of the local temperature. Collected data are sent to controllers (denoted by *C*), which send commands to actuators (denoted by *A*), e.g., coolers. Controllers can be either fixed (e.g., on the wall of each room) or embedded into mobile devices to let people set the preferred temperature of the environment around them. A centralized controller is also present to perform some global operations on the temperature control system, e.g., to ensure that room settings comply with the total energy budget.

To properly design this application, many aspects should be taken into account, e.g.:

- different concurrent tasks are involved, e.g., temperature sensing, and cooler activation;
- many instances of each task are present, e.g., a mobile controller for each user;
- tasks are hosted by physical devices with different capabilities, e.g., mobile vs. fixed embedded systems;
- physical channels can be either wireless or wired;
- communication protocols can be either reliable or un-reliable, best effort or with priority;
- the position in which sensors, controllers and actuator are placed affects the communications among them, the sensing performance and the control policy.

In traditional embedded systems applications the focus is the correct behavior of the single system while in distributed embedded systems applications the focus is the *good behavior of the global distributed application*, which depends not only on nodes but also on the communication infrastructure among them. For instance, in a traditional cellphone application, the objective is to guarantee a good communication between two users. This is assured by using techniques and protocols which are unaware of the usage environment, as the presence of other users in the same area, since the transmission time and

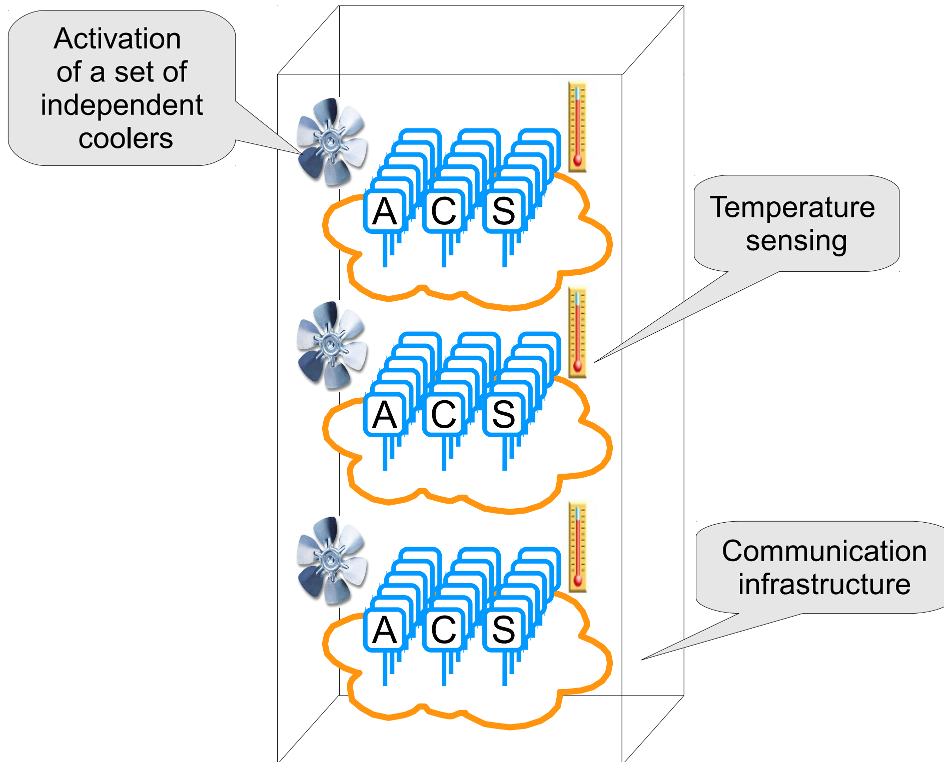


Fig. 1.1. Example of application for building automation.

bandwidth is divided into pre-fixed slots. Thus, traditional cellphones are point-to-point applications. Conversely, it is possible to implement cellphones which perform Quality-of-Service (QoS), exploiting the environmental informations, like the presence of other cellphones, to choose the transmission bandwidth and time. Such QoS-oriented cellphone applications represents a distributed application.

Moreover, in some highly-cooperative applications it could not matter if a single node does not work correctly, as long as the global application behavior respects design objectives. For instance, in a temperature monitoring application, the objective is to have a good estimation of the average temperature, and not that each single node will work correctly.

Therefore, as depicted in Figure 1.2, in networked embedded applications the traditional flows of design, verification and dependability analysis should be applied not only to each node in isolation but also to the whole distributed systems. Thus, distributed applications pose new questions to designers, traditionally mainly interested in the specification of each embedded system in isolation. Some issues are:

- Calculate the required number of nodes.
- Estimate the maximum number of supported nodes.
- Discover the best assignment between tasks and network nodes by taking into account tasks' requirements and nodes' capabilities.

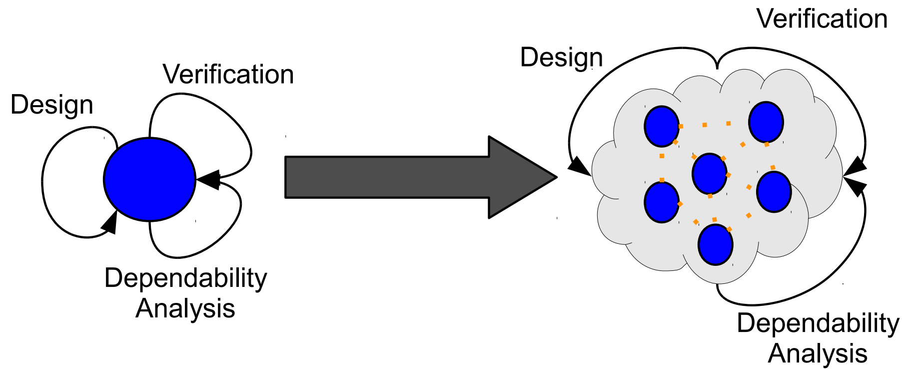


Fig. 1.2. Design focus shift: from single node to global behavior.

- Given a partition of the environment into zones, find out the best task-to-zone assignment.
- Given an assignment of tasks (with the corresponding communication requirements) to nodes, decide which channel types and protocols should be used among them, to satisfy such requirements.
- Decide if it is useful to introduce additional intermediate systems (e.g., routers, range extenders), to improve communication performance.

Solving these issues leads to the *complete definition of both the nodes and communication infrastructure*. All these new questions are strictly related with the communication aspects of the application, and hence, it is needed a design process which considers not only HW and SW, but also the Network as a fully-qualified design dimension.

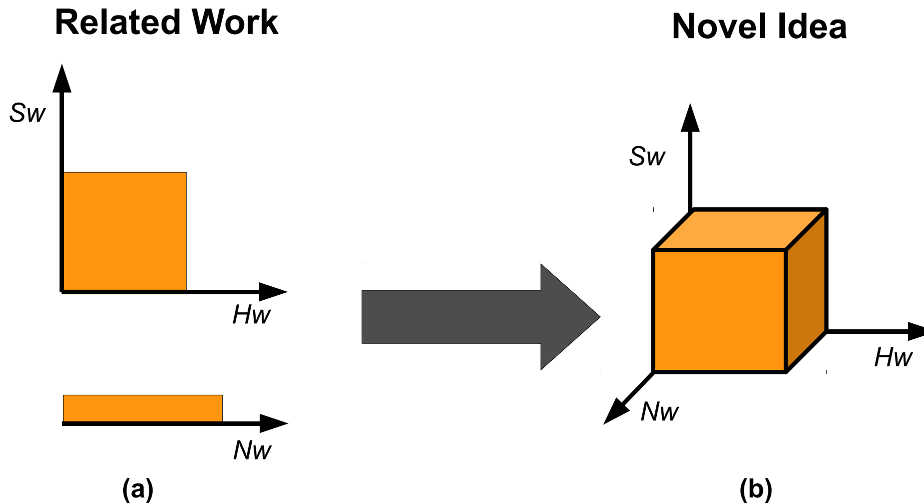


Fig. 1.3. The key idea: a design methodology which considers HW SW and NW design space dimensions.

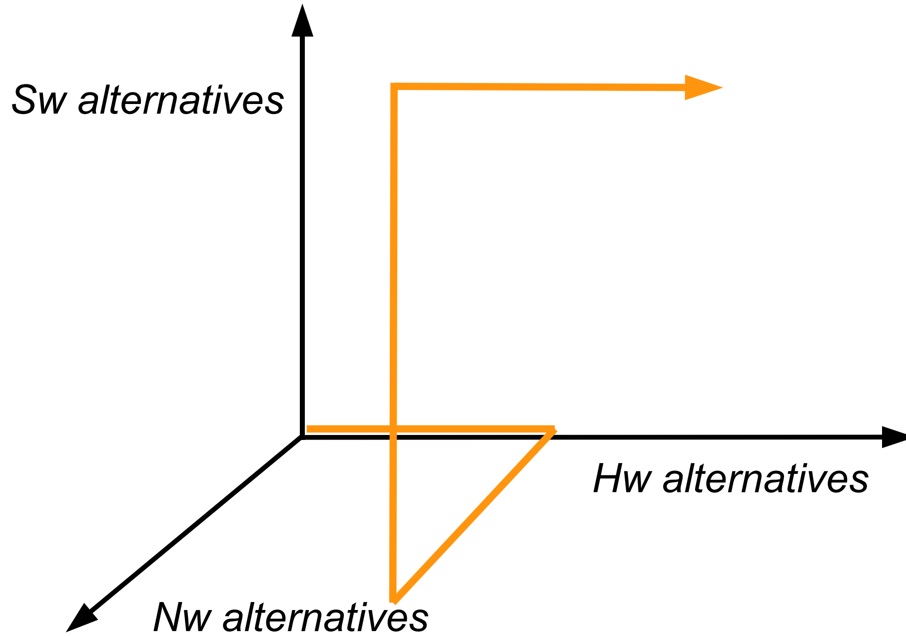


Fig. 1.4. Three-dimension design space of networked embedded systems.

As reported in Figure 1.3.a, the related work on the design of networked systems can be divided into two categories (Section 4.1):

1. focused on HW/SW design and not considering the network as part of the design space;
2. focused on network-related problems, but without a general design methodology taking into account the whole application.

The novel idea followed by this thesis is to try to merge these two opposite approaches into a single design space (Figure 1.3.b), which will allow the codesign of HW, SW and Network.

In fact, the lack of network modeling may lead to non-optimal solutions in the system design, since recent work demonstrated that HW/SW design and network design are correlated [119]. Choices taken during the System design exploration may influence the network configuration and viceversa. The result is a three-dimension design space as depicted in Figure 1.4. The vertical and horizontal dimensions address both the refinement of the System model and the optimization of its algorithms, i.e. it addresses the HW and SW design dimensions. During this process the network model is used both to drive architecture exploration and to verify that communication requirements are met. The depth dimension represents the design space of the network model in which the choice of protocols, channel and quality-of-service techniques may impact on the design of the networked embedded systems.

This thesis presents a global design flow, which starts from application specification and leads to the actual distributed application throw refinement steps involving HW, SW, and Network components of the distributed embedded system. Its applicability regards

all the distributed applications where the network is not fixed a priori, as in the case of wireless sensor networks and networked control system; in the final chapter of the thesis some remarks will introduce the use of this approach also in the context of networks on chip.

Similarly to the definition of components in electronic system design, the process of network dimension design is referred with the name of *network synthesis*; it consists in the following steps:

1. specification of the communication requirements of the application to be designed;
2. progressive refinement of the specification through design-space exploration;
3. mapping of the solution onto actual objects, i.e., physical channels, protocols, intermediate systems.

Moreover, this thesis addresses also some generic issues related to NES's design, such as design translation between different HDL's, dependability analysis, and system/network simulation.

1.1 Thesis structure

This work is structured as follows. Chapter 2 introduces some basic concepts useful to better understand the topics addressed in this thesis. The global design flow is described in Chapter 3, which is made by many sub-methodologies and supported by the corresponding tools. Each involved methodology and tool is described in a dedicated chapter (4 5 6 7 8 9 10), which includes a detailed analysis of the related state of the art, a deep description of the methodology or tool, and experimental results. Chapter 11 reports a complete application of the proposed design flow. Finally, conclusions and remarks on possible extensions of the work are drawn in Chapter 12.

2

Background

This chapter briefly introduces some basic concepts related to the topics addressed in this thesis. The key ideas are reported to better understand subsequent chapters. For a more detailed description about the topics here introduced, please refer to the bibliografy. State-of-the-art analysis regarding each research topic of the thesis is reported at the beginning of each related chapter.

Section 2.1 outlines some areas in which NES's are adopted, and some common communication protocols. Section 2.2 describes concepts relative to the System Level Design, like standard design flows for embedded systems, and the most used design languages. Finally, verification issues and techniques are briefly reported Section 2.3.

2.1 Networked Embedded Systems

Networked Embedded Systems (NES's) are special-purpose computation devices in which the communication aspects play a fundamental role; they are adopted in many different fields, e.g., set-top boxes, mobile terminals, process automation, home automation, automotive.

A NES category which is becoming of common usage in our houses is constituted by *set-top boxes* [128]. Set-top boxes are devices that connect a television and an external signal source, converting the signal into television content and vice versa. Common set-top boxes are satellite decoders and IPTV boxes. This kind of NES's have to be cheap, since they are consumer-oriented. Communication aspects are ever more important for these kind of devices to increase the delivery quality of multimedia content (e.g., through priority transmission and packet protection) and to introduce security guarantees (e.g., through cryptographic protocols).

Mobile terminals [132, 166] are another wide area of NES's. The most common mobile terminals are cell phones. In the last years, cell phones have notably increased their complexity, thus, they are becoming more similar to general purpose computers. In fact, they are currently able to run many different applications, and they integrate also many devices like cameras and sensors. Commonly, mobile terminals must have low cost, small size and low power consumption; they will support the ever growing wireless capacity to provide richer services to the users. Finally, security is an issue due to the transmission of

sensible data, and user interfaces must be of easy usage, more intuitive than a traditional PC.

Home automation [91,148] is another consumer-oriented topic, which involves NES's. A typical example in this context has been described in the introduction of this thesis and in Figure 1.1. This kind of applications can have constraints on dependability and real-timeness, to assure quality of service, or when damage to people is possible. Power consumption is usually a tight constraints, to reduce home maintainance costs. Finally, since they are consumer-oriented applications, they must be cheap and possibly miniaturized.

In industrial context, NES's are used for *controlling* [29] and *automotive* [99, 102, 121]. Such NES's can have very hard constraints on realtime operation and dependability, because of the safety-critical application context. For instance, the temperature-monitoring of an oil plant requires a very dependable infrastructure, with a quick activation of health-safety systems in case of danger. Moreover, there can be security issues, when wireless transmissions or Internet connections are adopted.

Many of the reported NES applications fall into the category of *Wireless Sensor Networks* (WSN's) [9]. WSN design constraints vary depending on their usage, but the most common are low power consumption, miniaturization, low computation capabilities, low communication rate, and high degree of dependability.

2.1.1 Communication protocols

There are many communication protocols used to interconnect NES's.

IEEE 802.11 [95] is a set of standards concerning Wireless Local Area Networks (WLAN's). Each 802.11 standard has been derived from the original 802.11-1997 standard as amendment. Most common protocols belonging to this family are 802.11a, 802.11b, 802.11g, 802.11e, and 802.11n.

The 802.11a standard uses the same data link layer protocol and frame format as the original standard, but it has an orthogonal frequency-division multiplexing physical layer, which allow to have a maximum bitrate of 54 Mb/s. It has been released in 1999.

In the same year, another amendment was released, namely the 802.11b. It it uses the same channel access method of 802.11-1997, but it implements an extended modulation technique leading to a maximum bitrate of 11 Mb/s.

802.11g has been released in 2003. It uses the same band of 802.11b, but integrates the transmission schema of 802.11a. Thus, it can have a maximum bitrate of 54 Mb/s, as 802.11a.

802.11e defines QoS enhancements through modification to the Media Access Control (MAC) layer. Thus, it is an important improvement for applications which are sensible to delays, such as multimedia streaming.

In 2009, IEEE has released the 802.11n amendment, which allows bitrates from 54 Mb/s to 600 Mb/s, through the use of multiple-input multiple-output antennas.

Another important standard is the 802.15.4 [111]. Released in 2006, it specifies the physical and MAC layers for low-rate wireless personal area networks. It allows a maximum bitrate of 250 Kb/s. For managing packet collisions, it uses the Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) schema. Supported network topologies are star and peer-to-peer.

ZigBee [171] is a protocol which adopts the 802.15.4 for its lower levels. It is a specification of high-level communication facilities for small and low-power digital radios. For

this reason, it is widely adopted for home automation and entertainment, and for wireless sensor networks. The ZigBee standard is not published by IEEE, since it is maintained by the ZigBee Alliance. Any commercial implementation requires the payment of fees to join the ZigBee Alliance, while the standard is freely available for non-commercial purposes.

Fieldbus [64] is a family of industrial wired network protocols, used for realtime distributed control. It is standardized as IEC 61158. The standard encompasses the physical, data link and application layer. Standards belonging to the Fieldbus family are not directly interchangeable.

One of the main competitor of Fieldbus is Controller Area Network (CAN) [43], defined in the ISO 11898 standard. CAN is designed to allow microcontrollers and devices to communicate each other without a host computer. CAN is a multi-master broadcast serial bus, characterized by the capability of each node to send and receive packets. One of the main application fields of CAN is automotive to connect devices, controllers and sensors in today's cars.

802.3 [96] is a collection of IEEE standards defining the physical and MAC layers of wired Ethernet networks. It is related to Local Area Networks (LAN's), but it also has some wide area network applications. The maximum bitrate is strictly related to the physical channel type and to the actual standard, with the maximum bitrate of 100 Gb/s, as implemented in the 802.3ba standard.

2.2 System Level Design

System Level Design regards methodologies and languages suitable to design systems at high level of abstraction.

A possible analogy to explain System Level Design issues, is the evolution of languages in the SW field. SW languages have evolved from plain assembly to more complex and abstract languages like C++, Java, Ruby, which provide high level data manipulation and advanced programming features like objects, generics, polymorphism, reflexion and closures. These languages let to create in a shorter amount of time programs which are more complex but at the same time easier to maintain and improve. Moreover, the top-down code generation. i.e. the generation of machine code from high level languages, has been supported by the implementation of automatic tools, i.e. compilers, which allow to avoid error-prone and time consuming manual conversion.

The same event has happened in the fields of embedded systems and HW design. Raising the level of abstraction has required the introduction of new design languages and tools.

This section introduces some key concepts about the System Level Design. Section 2.2.1 explains the terminology and the key ideas about usual top-down design flows. Languages used to design embedded systems are similar to SW programming languages, but they have also some important peculiarities. Their most important features are reported in Section 2.2.2 Finally, Section 2.2.3 explains a design methodology, which is focused on the adoption of a middleware as a new design space dimension.

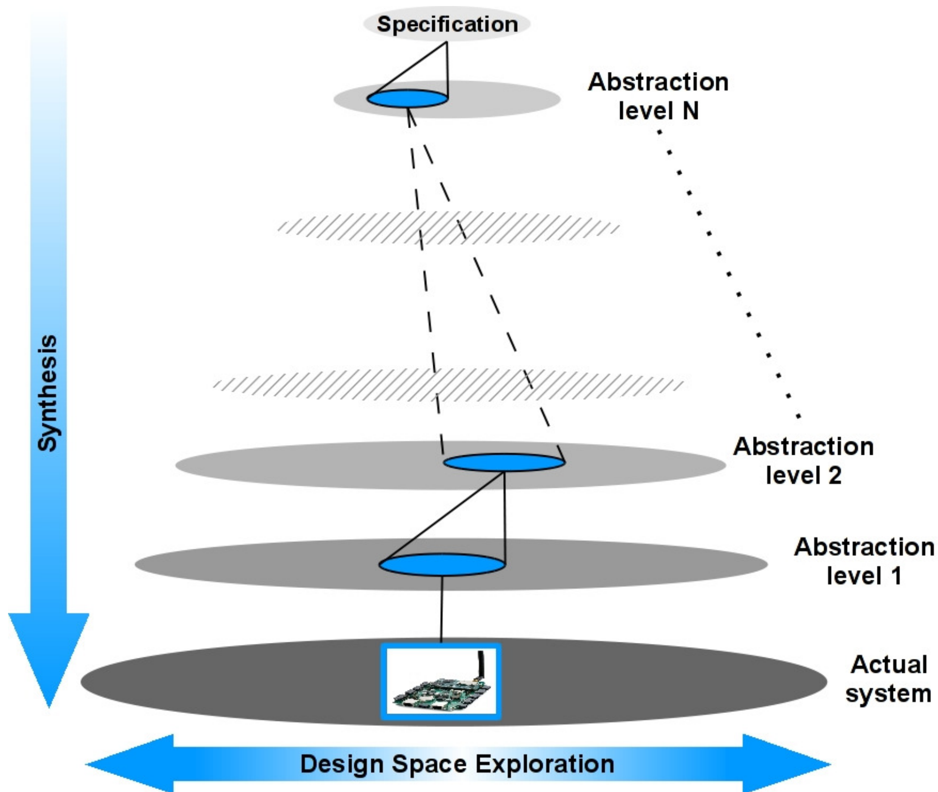


Fig. 2.1. Schema of a top-down design methodology.

2.2.1 Top-down design flow

Figure 2.1 illustrates the schema of a typical top-down design methodology. The design flow starts with high level specification, that can be given in natural language, in a specification language like UML [125] or MARTE [123], or through a functional description given in an implementation language, like C++ or SystemC.

Each design step consists in refining the model, i.e. adding implementation details, and thus, moving the application model from a more abstract to a more concrete representation, until the actual system is completely designed. This process is known as *synthesis*. After each refinement step, the application model shall be verified, in order to guarantee that all the requirements and constraints are satisfied, and that eventual implementation bugs are fixed. In order to simplify the synthesis and to reduce the time-to-market, many research works are focused on providing methodologies and tools able to automatically synthesize from high abstraction levels.

Usually, during each synthesis phase it is possible to take different design choices, i.e., to have many candidate solutions on which map the application model. For instance, the specification could describe the network communication protocol in terms of general properties and requirements, such as the desired level of reliability in the communication. On the other hand, during the synthesis, an actual protocol shall be chosen, and thus, different reliable protocols shall be evaluated to check which one is more suitable for the

application under design. The process to check different choices is named *Design Space Exploration* (DSE).

Synthesis and DSE can be seen as two strictly related topics, which lead to an optimization problem. For instance, in the HW field, designers want to synthesize chips w.r.t. an optimization metric, such as the minimization of the area or consumed power. This kind of problems are known to be NP-hard, and many research works aim at finding techniques on how to discover a *good*, but not always optimal, solution in a short amount of time.

2.2.2 Hardware Description Languages

Hardware Description Languages (HDL's) are languages used to design hardware systems. Usually, HDL's are able to describe systems at various abstraction levels, since typical design flows are based on top-down methodologies. Some common abstraction levels are:

- Behavioral: the functionalities are described by using high level operators. For instance, the sum of two integers is performed directly.
- Register Transfer Level (RTL): the circuit is described in terms of registers, and operations on signals and ports.
- Gate: the circuit is described in terms of logic ports interconnected by wires. The circuit is usually referred as *netlist*.

HDL's have their semantics specified in terms of an event-driven simulation behavior. This simplifies the implementation of actual simulators suitable to check the system correctness. As functions are the basic structures of software programs, the basic structures of an hardware model are *processes*, which have a *sensitivity list*. Each time a port or signal belonging to the sensitivity list changes its value, the process is executed. Processes are grouped into *modules*, and modules can be encapsulated into other modules, to create complex hierarchies, similarly to what can be done with objects in software programming languages. For instance, the model of a computer could contain various sub-modules, one for each HW components, like CPU, RAM, bus, etc.. Since all the processes sensitive to the same signal will be activated at the same time, the simulation semantics specify that processes will be executed in parallel and in a unpredictable order, but each process will execute atomically w.r.t. shared variables.

Besides software-like types as integers and booleans, HDL's have special types for data, which are strictly related to hardware systems. Typical types are bits and logics. Logics are bit-like types which can have `'0'` and `'1'` as values, plus some values to describe special behaviors. For instance, the value `'X'` denotes that the value of the logic bit is unknown.

Next paragraphs introduce some common HDL's.

Verilog

Verilog [52] was introduced in 1985 by Gateway Design System Corporation, as a proprietary language. In 1995 it became an IEEE standard (IEEE 1364) and a revision has been published in 2001. The syntax is case sensitive and similar to `C` language. The Verilog standardization working group is no longer active, since its activities have been taken

over by the IEEE P1800 standards group, which is working on standardization of System Verilog as well as taking on maintenance ownership of IEEE 1364. Despite this fact, currently Verilog is one of the most adopted HDL's.

The primary types are registers (`reg`) and wires (`wire`). Registers store values like normal variables, whereas wires do not store values, since they represent physical interconnections between structural entities such as gates.

VHDL

The VHSIC Hardware Description Language (VHDL) [94] arose out of the United States government's Very High Speed Integrated Circuits (VHSIC) program, initiated in 1980. VHDL became the IEEE 1076 standard in 1987 and a revisioned version has been published in 1993. It is a case-insensitive strongly typed language, and it has a syntax similar to ADA.

A specific feature of VHDL is the clear separation between module interfaces and their implementations. Designers can create many implementations of the same interface, choosing at compile time which one shall be used. This is useful to design modules at various abstraction levels, without changing module interconnections.

VHDL supports generic programming on constant values, which shall be defined at module instantiation time.

SystemC

SystemC [97] is a recently introduced HDL, which represents an extension of C++. It has been created by the Open SystemC Initiative (OSCI), and in 2005 it became the IEEE P1666 standard. A reference simulator is freely available from the OSCI site (www.systemc.org).

Due its C++ base, SystemC has a great flexibility and supports advanced features like templates. Hence, SystemC has a more complex syntax than other HDL's, and it has also the same semantics lacks of C++.

An important extension to the RTL set of primitives is the Transaction Level Modeling (TLM) [159], which allows to describe designs at Electronic System Level (ESL). At ESL, the communication between modules is separated from their functionalities. Communication infrastructure like buses and FIFO's are modeled as channels, with standard interfaces. This hides communication details to modules, and thus, designers can check different communication schemes without changing module implementation. TLM simulations are usually two order faster than functionally equivalent RTL simulations.

Thanks to this high level of abstraction, TLM is suitable to describe systems with HW & SW mixed components, before performing usual HW/SW partitioning.

2.2.3 Abstract Middleware Environment

The *Abstract Middleware Environment* (AME) [68] is a design flow based on the assumption that a middleware will be present in the final distributed application. It is introduced as a background notion since it is based on three abstraction levels, which can be taken as reference to develop a more general design and verification methodology, as proposed in this thesis (see Chapter 3).

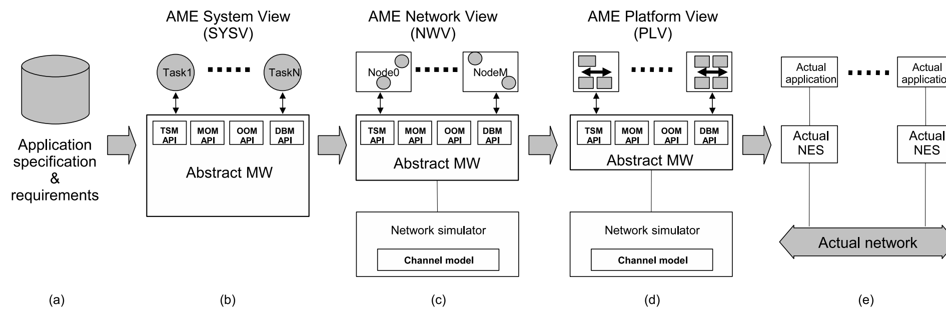


Fig. 2.2. AME-based design flow.

AME simulation environment contains an abstract model of the middleware supporting different programming paradigms available in literature, i.e. Object Oriented (OOM), Message Oriented (MOM), Tuple-Space (TSM) and Database (DBM) paradigms. AME has been introduced to avoid early assumptions on middleware characteristics, and to allow a parallel design of the application and of the other parts of the system, like the network. Thus, with AME the middleware becomes a true project dimension, which requires DSE and synthesis (i.e. the mapping on a real middleware). AME allows also the reusing of application code thanks to automatic conversion between different middleware programming paradigms [70].

The design flow supported by AME is depicted in Figure 2.2; it consists of three steps.

In the first step, namely System View (SYSV) (Fig. 2.2.b), the application is modeled in SystemC, following the functional requirements (Fig. 2.2.a). The application consists of different tasks using the middleware primitives for communications, but actually the simulation does not take into account the presence of the network. In the next step (Fig. 2.2.c), namely Network View (NWV), tasks are grouped into network nodes, and the abstract middleware is binded to a network simulator, thus considering network effects on communications between tasks belonging to different nodes [69]. Next, at Platform View (PLV) (Fig. 2.2.d), HW/SW partitioning is performed on each node as currently done for traditional embedded systems. The simulation is carried on by using cosimulation of HW, SW and network components. Finally, the mapping on actual nodes leads to the complete implementation of the system, where the abstract middleware is replaced by the actual middleware (Fig. 2.2.e).

2.3 Assessing embedded systems correctness

Checking the correctness of embedded systems is a fundamental step during design phases, in order to assure the quality of the final product. Designers can be interested in assuring at least three different properties:

- The system implementation is bug-free. This goes under the general term of *verification* [98].
- The system respects the specification constraints and objective. This is named *validation* [98].
- The system is able to work properly even in case of some errors. This issue is named *dependability* [13].

In literature there are two main research areas devoted to provide methodologies to assure such properties. The first one is represented by formal methods, and in particular by model checking, the second one relies on testing.

Formal methods are based on mathematical models and abstract representation of the actual system. Thus, they are able to assess system correctness. Unfortunately, formal methods use NP-hard algorithms, and thus, they are usually adopted to check only important portions of the whole system.

Testing is based on the simulation of the system. As such, testing cannot give a final answer about system correctness as formal methods, but it can be adopted also with very huge designs. Since testing is based on simulation, it becomes crucial which inputs are given to stimulate the design. A sequence of inputs is named *testbench*, and a set of testbenches is named *testset*. A testset is considered of high quality if it is able to simulate the design achieving a high score w.r.t. a given metric.

The following section illustrates the Fault Injection technique, which can be adopted both to generate high quality testsets and to assess the dependability of the system.

2.3.1 Fault Injection and Fault Simulation

Fault injection is a methodology widely adopted in many different fields. A system *failure* is an even that occurs when the system behavior deviates from the correct service. An *error* is the system part which may cause a subsequent failure. Actual systems may fail due to errors in the implementation or in the specifics. A *fault* is the cause of an error [13]. Fault injection is based on the concept of fault model, i.e. a set of mutations, which represents an abstraction of actual faults of the system. The most common fault injection technique, used to inject the Design Under Verification (DUV), consists in the mutation of the DUV source code; for instance, an addition operation can be substituted with a subtraction. In literature there are many fault models, each one tailored to actual contexts and able to reproduce different errors. For instance, a fault model used at gate level is the Stuck-at, which forces a bit to hold always one or zero. This fault model can represent the abstraction of a broken wire.

Fault injection can be used at least for two objectives:

1. As a coverage metric, namely Fault Coverage.
2. As a dependability analysis technique.

The fault coverage metric measures the number of injected faults that originate a difference in the result of a computation. These faults are called *propagated* or *detected*. The fault propagation is usually checked by simulating the injected design. Ideally, a good testset should propagate all the faults, but in practice an high percentage of propagated faults can suffice. Fault coverage is then used to perform the *testbanch qualification*, i.e. to check testbenches quality. If the required minimum percentage of propagated faults is not reached, the testset shall be enriched with new tests in order to increase the fault coverage. On the other hand, if a test is unable to propagate any fault, such a test is considered to stimulate too weakly the design, and hence it can be removed from the testset.

Considering the dependability analysis, a system is defined *dependable* if the behavior is correct even if an error occurs. Thus, a fault can be injected to check if the system behaves correctly even in case of that fault. If the fault propagates, the system is not dependable w.r.t. that fault, and thus, it shall be modified to increase its dependability level.

3

Proposed design & verification methodology

This chapter describes the proposed design and verification methodology for Networked Embedded Systems (NES's). Such a methodology is constituted by sub-methodologies to solve specific problems, and it is supported by some tools. All the presented sub-methodologies and tools have been developed during these years of Ph.D. studies, and thus, the presented global flow represents the summary of all the research work. The design flow is described in Section 3.1, and a related example is reported in Section 3.2. The details of sub-methodologies and tools are explained in the following chapters of this thesis.

3.1 Methodology overview

Figure 3.1 shows the proposed design flow for NES's, which starting from high-level specification leads to the actual NES via three macro-phases, each one addressing a different design aspect. Greu rounded boxes are the threee macro-phases on which the proposed design flow is based. Orange ellipses represent design methodologies integrated into the proposed design flow, while yellow boxes represent the developed tools. Remaining grey ellipses represent general design issues. For each block, the corresponding thesis chapter is reported.

The input of the design flow is the application specification, which can be given in SystemC, or in a specification modeling language such as UML [125] or SysML [124].

The first phase is named *System View* (SYSV) and implies the implementation of an application model in SystemC. At this level of abstraction, application functionalities are interconnected directly, and thus, they can interact in the same way a non-distributed application can do. The objective of this phase is to create an application simulable model, in order to get some initial feedbacks about design choices. Application functionalities are partitioned into tasks, which can be implemented exploiting architectural patterns. Thus, designers are able to check if it is required to change involved tasks and their interactions. Moreover, some high-level verification steps can be performed, to check the satisfaction of application requirements and the correctness of implemented code.

The second phase is named *Network View* (NWV). The network, which was abstracted at SYSV, is made explicit, and the tasks are grouped into network nodes. This implies that the simulation must consider eventual communication delays due to the network.

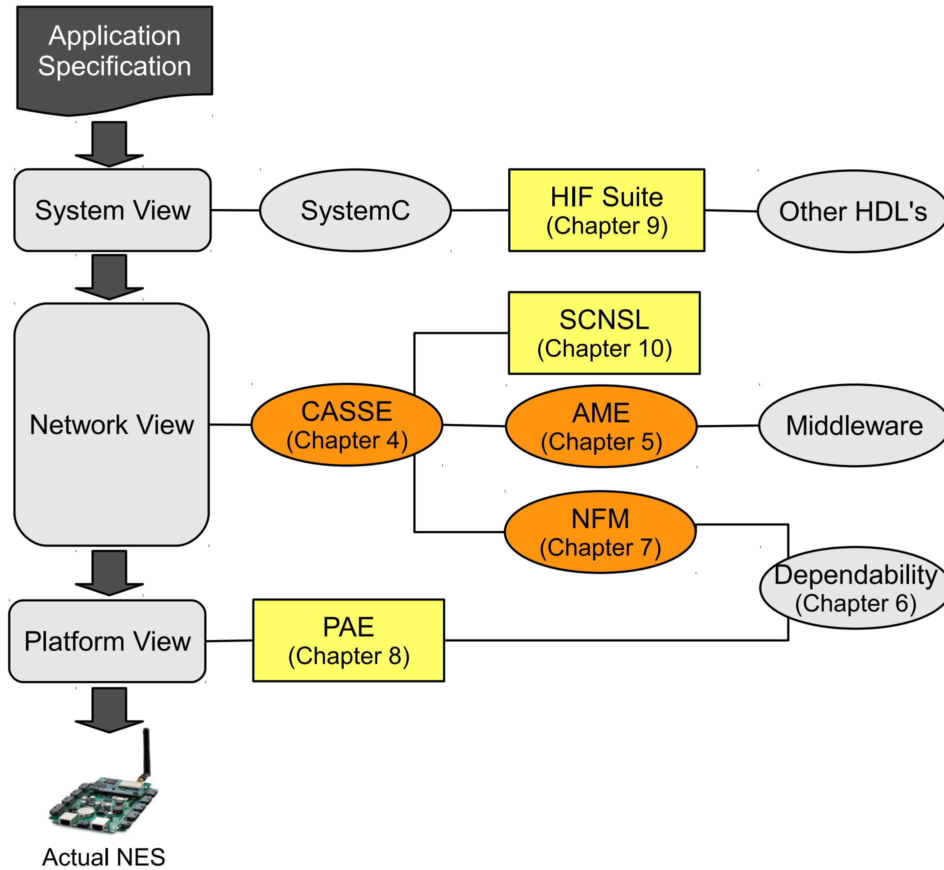


Fig. 3.1. The proposed design flow for distributed systems, with support methodologies and tools.

During this phase, the network design space shall be completely designed and verified. For instance, designers shall choose the number of nodes to deploy, which tasks to assign to a node, where to place the nodes, the communication protocols, the kind of communication channels (wired vs wireless), etc..

The last phase is the *Platform View* (PLV). This phase is performed on per-node bases, and targets the system portion of the application, i.e. the HW and SW design space dimensions. Inside each node, HW and SW are co-designed, co-verified, partitioned and synthesized. For instance, a TLM HW model can be refined at RTL, in order to have better performances and power informations. Since the Network has been completely designed, during this phase all the state-of-the-art design methodologies that address only HW and SW can be reused.

The output of the PLV phase is the actual NES.

Since the proposed design flow is based on SystemC, to allow reuse of components written with other HDL's, and to integrate HDL tools which do not support SystemC, the the proposed framework integrates a suite of translators between the various HDL's. Such a suite is named *HIF Suite* (Chapter 9) [22, 23].

At NWV, the network design space exploration and synthesis should be performed exploiting techniques that addresses explicitally this design dimension as their main design goal. The *Communication Aware Specification and Synthesis Environment* (CASSE) has been proposed to solve this issue. The CASSE methodology is based on a formal model of NES entities (Chapter 4) [75], which captures application, communication and computation requirements from a communication point of view. The formalization of the NES leads to the formulation of an optimization problem, which can be then solved using both analytics and simulative state-of-the-art techniques.

The simulation at NWV of the application model is useful to perform both design space exploration of the network, and verification of the implemented application. The simulation requires the integration of a network simulator with SystemC. While in related works the co-simulation of NW and HDL's has been adopted, in this thesis the effort has been to implement a network simulator directly in SystemC, in order to avoid performances overheads. The implemented simulator has been named *SystemC Network Simulation Library* (SCNSL) (Chapter 10) [47, 73, 77].

NES's are very complex systems since both system and network portion of the application must be accurately designed. These issues can be simplified by adopting a middleware, since they allow to simplify tasks communication and to abstract hardware details. On the other hand, the presence of a middleware creates new design issues at NWV, since middleware communication and computation overheads shall be taken into account during the network DSE. For this reason, CASSE has been extended and integrated with an already existent methodology, namely the *Abstract Middleware Environment* (AME) [68–70] (Chapter 5) [71]. AME is a middleware-centric design methodology which considers the middleware as a new design space dimension. Since AME is written in SystemC and it is based on the same three macro-phases of the proposed global design flow, i.e. SYSV, NWV and PLV, its integration has been quite straightforward.

Wrong design or malfunction of distributed systems can lead to drastic performance degradation, severe damage to people and to the environment, and significant economic losses. For this reason, the standardization of procedures to improve safety has become an issue in the general context of embedded systems [41], and in the specific context of avionics [62] and automotive [99].

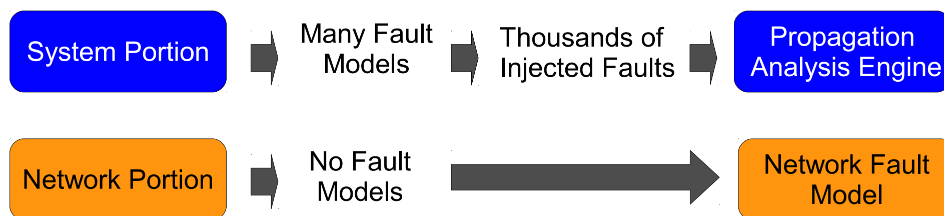


Fig. 3.2. Dependability issues and thesis contributions.

Among all possible safety improving methodologies, *fault injection* has been chosen, since it is widely adopted in industrial context. Fault injection can be used to both perform dependability analysis and testbench qualification. Its applicability depends on two requirements: a fault model suitable for the application context, and a fast fault simulation (Figure 3.2).

In the proposed design flow, at NWV it is required to be able to specify dependability constraints related to the network dimension, by exploiting the CASSE formal model. Thus, CASSE has been extended with parameters able to express the dependability of each network-related entity (Chapter 6) [76]. Then, it must be checked if dependability requirements of the candidate solution have been met.

At this level of abstraction, a suitable fault model shall target explicitally the communication. Since in literature there was not such a kind of fault model, the *Network Fault Model* (NFM – Chapter 7) [72, 74] has been introduced. The NFM has been derived by classifying all possible communication errors of a wireless channel represented as an EFSM, thus assuring that all incorrect behaviors were reproduced, and that meaningless mutations were excluded.

Regarding the system portion of the application, fault injection can be performed on single components at PLV by injecting functional faults taken from the literature, like Bit-Coverage [63]. Since such fault models usually inject thousands of fault instances, it is important to optimize fault simulation performances. A methodology to perform parallel functional fault simulation has been implemented in a tool, named *Propagation Analysis Engine* (PAE – Chapter 8) [49]. The key idea of PAE has been to extend to RTL parallel fault simulation techniques typical of the gate level.

In the remaining of this thesis, all these methodologies and tools are described, addressing their key ideas and issues. Their validity is proven by the reported case studies.

3.2 Example of design flow adoption

This section reports an example which is an overview of the adoption of the proposed design flow. As reference application, let us consider the design of the temperature monitoring application described in the introduction of this thesis, and depicted in Figure 1.1.

The input of the design flow is the application specification, which reports the functional and dependability requirements of the application under design. For example, a functional requirement may assert that the temperature of each room shall automatically adapt to user's preferences specified through mobile terminals. An example of dependability requirement is to assure a good Quality of Service even in case of some temperature senror breaking. Other common requirements may be the minimization of power consumption and of the system cost.

This application could require the design of a custom temperature sensor node with low power consumption. Designers could have already implemented a suitable sensor in previous projects, and they would like to reuse it. For example, the sensor could be written in VHDL. Since the proposed design flow is based on SystemC, they can *automatically* translate the sensor design by using the HIF Suite.

The first design phase is to implement the application functionalities in SystemC at SYSV. For this purpose, in literature there are works relative to Model-Driven Design, which automatically translate application specification to actual code [11, 137–139]. Another chance is a manual step of implementation.

After the implementation at SYSV, designers can perform verification to assure conformance to the specification, and to fix eventual bugs. Verification can be accomplished through simulation, since the design is simulable by using the standard SystemC distribution, or exploiting other verification tools.

The next design phase, namely the NWV, consists in the refinement of the network portion of the application. In this design step, designers shall choose communication protocols, node positions, task-to-node assignments, etc. To support this complex phase, designers can adopt the CASSE methodology. To perform the design space exploration and the verification through simulation, the SystemC platform must be interconnected with a network simulator, since it is important to check the communication performances. The network simulation can be performed by using SCNSL.

To verify that the network portion satisfies the dependability requirements, the design is injected with the Network Fault Model. Some important feedback can be collected during this phase. If dependability requirements are not reached, different protocols and node positions could be explored. For example, the sensor nodes can have a failure percentage reported in their data sheet. Thus, by using the NFM it is possible to discover the minimum required number of nodes to guarantee a good temperature estimation even in case of failure.

At PLV, the remaining design steps are on per-node basis. For instance, different SW/HW partitioning can be evaluated and the HW portion will be refined at RTL and then synthesized. Fault Injection can adopted to both testbench qualification and dependability analysis. To speed up the fault simulation of the HW portion, designers can exploit the parallel simulation techniques offered by PAE.

The result of this flow is the distributed application build over the corresponding networked embedded systems.

4

Communication Aware Specification and Synthesis Environment

This chapter presents a methodology useful to design the network dimension of a distributed embedded application, i.e. to perform the refinement at NWV. The methodology name is *Communication Aware Specification and Synthesis Environment* (CASSE), since it is based on a formal model and focuses on the network synthesis. Its applicability regards all the distributed applications where the network is not defined a priori, for instance, wireless sensor networks, networked embedded systems and networks on chip.

This chapter gives the mathematical basics of the network synthesis, in order to reconduct the synthesis to a traditional optimization problem. Exploiting such a formulation, traditional and state of the art techniques can be applied to solve the synthesis problem.

The main contributions of this work are:

- a modeling framework that allows to represent the communication aspects of the applications;
- an extended design methodology which considers the communication infrastructure as a dimension of the design space;
- case studies to show the potentiality and applicability of the approach.

The proposed methodology is also a first step towards the *computer-aided synthesis of the communication infrastructure*, i.e., the automatic choice of channel type, protocols and intermediate systems to support a distributed application. Synthesis is currently done for hardware components but, as far as we know, this is the first attempt to apply the same approach to the communication infrastructure.

The rest of this chapter is organized as follows. Related work is reported in Section 4.1. The CASSE design flow is introduced in Section 4.2. The proposed formulation of the problem is reported in Section 4.3. A first discussion of the synthesis methodology is provided in Section 4.4. The proposed methodology has been applied to some case studies. The objective is to clarify the explained concepts and the proposed formalization, and not to provide an example of complete workflow. The first case study (Section 4.5) regards a temperature-monitoring application. The second case study is an example of a distributed computing application (Section 4.6). Conclusions and future work are reported in Section 4.7.

4.1 Related Work

The synthesis of distributed systems with some degree of network design has been addressed by many research works, in different fields, such as wireless sensor networks (WSN). A virtual architecture has been proposed in order to simplify the synthesis of algorithms for WSN [15]. Some network informations, like the topology and high-level functionality, are used to configure the virtual architecture. However this work is mainly focused on the application part of the system rather than on communication aspects. Platform-Based Design (PBD) has been adopted to design WSN for industrial control [27]. As usual in PBD, the application is designed at high level and then mapped onto a set of possible actual candidates for the nodes. However, no guideline is provided about the selection of the appropriate network architecture and communication protocol. Scope-based techniques have been proposed in macroprogramming to specify complex interactions between heterogeneous nodes of a WSN [118]. However, the number of nodes and the network topology are an input of the technique, not a result as in the proposed approach.

A communication synthesis methodology and HW/SW integration for embedded system design has been addressed in [82]. The method is based on task graphs, and takes places after partitioning and scheduling, inverting the phases proposed by many other works, which perform scheduling assuming that the communication aspects have been already set. On one hand, this inversion has the advantage that the scheduling problem is simplified, since communication components will be designed later. On the other hand, it does not allow the reuse or integration of the methodologies which performs scheduling after network synthesis.

The design of Networks-on-Chips (NoC) offers an example of system-network integrated approach which is close to those proposed in this work. NoC's are embedded systems and, thus, they are designed with the traditional specification-refinement-synthesis flow; nevertheless they have also a communication infrastructure which is a simplified version of a packet-switched network. The internal NoC topology can be either irregular or have regular form, e.g., mesh, tour, ring, and butterfly [19]; its design presents problems similar to the one of traditional packet-based networks. For example, some research has suggested that the mesh topology is most efficient in terms of latency, power consumption and implementation simplicity [6]. Routing protocols are a design issue, too. In NoC's routing can be deterministic or adaptive according to the current state of the network and to appropriate quality-of-service (QoS) policies [6]. Regarding the latter, best effort and guaranteed throughput have been proposed [145]; they are very similar to those defined for TCP/IP [143]. The problem of the optimal mapping of tasks onto NoC's cores is known to be NP-hard. In some works, heuristics based on graph-decomposition techniques have been used [6,131]. A Mixed Integer Linear Programming (MILP) formulation of the problem has been proposed [37]. It assumes a regular 2D mesh topology, and shortest-path static routing. This methodology allows two different optimization criteria, i.e., minimization of the average hop distance (which is proportional to power consumption and communication delay), and minimization of the bandwidth (which consists in minimizing the most-congested link-queuing time, and maximizing the throughput).

Synthesis of communication protocols is another research topic related to the focus of this work. Automatic tools have been adopted to derive the actual implementation

of protocols specified through finite state machines [106, 170], Petri Nets [168], trace models [142], and languages like LOTOS [54].

A general modeling framwork for a global design flow could be useful to integrate the various contributes, in order to allow the NES design by exploring HW, SW and Network design space dimensions. This chapter presents a work in such a direction.

4.2 Design flow for network synthesis

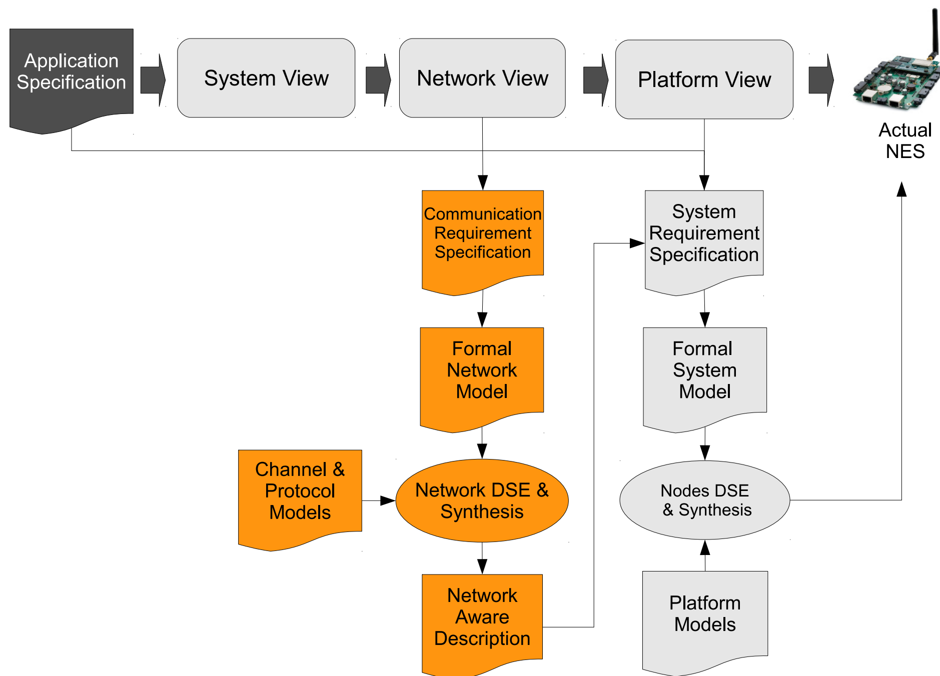


Fig. 4.1. The proposed design flow for distributed systems: new steps for network design (in orange) are added symmetrically to the state-of-the-art system design flow.

Figure 4.1 shows the design flow of a distributed embedded system; the right part of the Figure depicts the state-of-the-art design flow of embedded systems while the left part of the Figure introduces the proposed additional steps for the design of the network.

In the traditional system design flow a platform-independent model of the system is created starting from application requirements, both functional and non-functional. This is a pure specification model of concurrent resources (e.g., tasks) and their interaction through a communication medium (e.g., channels). Behavior in this specification is usually expressed through languages like UML and C/C++ or through the use of tools like Matlab/Simulink/Stateflow (please, refer to Section 4.2.1 for a survey).

This specification, together with a description of the target platform, is the subject of a Design Space Exploration (DSE) which maps tasks onto HW and SW components for

the target platform. The result is platform-dependent description of the system in which HW blocks are described by HDL languages like SystemC and VHDL while SW blocks are implemented in C/C++ and compiled for the target CPU.

This flow is perfect for isolated embedded systems, and it can be adopted at PLV, but in case of distributed applications made of many embedded systems it lacks a specific path devoted to the design of the communication infrastructure between them. For this reason, new steps are proposed as depicted in the left side of Figure 4.1, at NWV.

The new design path is quite symmetric with respect to the traditional design path since it applies the same concepts to the communication aspects of the whole system. Starting from the communication requirements, a *formal network model* is derived. It contains computational cost of each task, e.g., in terms of CPU and memory usage, and the requirements of communication flows between tasks, e.g., their throughput and permitted latency. The output of this phase will be a parametric model with free and bounded parameters and constraints among them.

This formal model of the network, together with a description of actual channels and protocols, is the subject of design-space exploration aiming at searching the optimal solutions of the parametric model obtained in the previous phase. Actual channels and protocols are chosen according to their affinity to the optimal solutions. This last phase is named *network synthesis*. The final result is a *network-aware description of the application* with the mapping of application tasks onto network nodes, their spatial displacement, the type of channels and protocols among them, and the network topology which may provide additional intermediate nodes to improve communication performance.

The network-aware description of the application contains important information for the design of each node of the network, i.e., the list of tasks assigned to it and the presence of new tasks to handle network protocols. For this reason, this description is used as input in the traditional design-space exploration of each node as reported in the right part of Figure 4.1.

4.2.1 High-level system specification languages

The elements of the formal network model depicted in Figure 4.1 can be extracted from a high-level description of the application created by well-known languages as those reported in this Section.

MARTE [123] is a profile of UML [125] designed to allow an easy specification of real-time and embedded systems. It provides some sub-profiles, like Non-Functional-Properties (NFP), which allows to describe the “fitness” of the system behavior (e.g. performance, memory usage, power consumption, etc.). The Software Resource Modeling (SRM) and the Hardware Resource Modeling (HRM) profiles are derived from NFP, and they address the modeling of resources.

The System Modeling Language (SysML) [124] is an extension of UML to provide a general-purpose modeling language for systems engineering applications.

Mathworks has developed Simulink [156] and Stateflow [157] to model and simulate dynamic and embedded systems. Simulink models an application as the inter-connection of dynamic blocks (e.g., a filter) or digital blocks. Stateflow describes applications as finite-state machines. The tools can also be combined to represent hybrid automata.

The Ptolemy Project [34] is born to model concurrent real-time and embedded systems. One of its main advantages is the support for heterogeneous mixtures of computation models. Ptolemy supports simulation by using the actor-oriented design; actors

are software components executed concurrently and able to communicate by sending messages throw interconnected ports. Ptolemy also supports communication modeling through Khan process networks [103]. Khan Process Networks (KPN) are a distributed model of computation based on tokens. The focus of KPN's is on the flow of computation and thus, KPN seems well suited to check properties on the communication schema.

SystemC [97], initially born as hardware description language, has been extended with the Transaction Level Modeling (TLM) [159], to describe HW/SW systems. SystemC and TLM allow to describe tasks as nested components with event-driven or clock-driven processes. Communications between tasks can be described by using standard protocols and payloads which simplify the specification of their behavior. TLM has born to represent local communications, such as bus interconnections or accesses to devices.

SpecC [33] is an extension of the C language to be used as system-level design language, like SystemC/TLM. The SpecC methodology is a top-down design flow, with four well-defined levels of abstraction. It allows different ways to describe the target control (sequential, FSM, parallel and behavioral). One key concept of SpecC is the clear separation of the communication and computation model which can be useful to specify computation and communication aspects of tasks.

Metropolis [32] is a framework based on the idea of meta-model to support various communication and computation semantics in a uniform way. This approach implements the abstract semantics of process networks and uses the concepts advocated by the platform-based design methodology, i.e., functionality and architecture across models of computation and abstraction levels, and the mapping relationships between them.

The proposed formal model is not a model of computation as TLM or KPN's, since its focus is on the specification of requirements. Moreover, it is network-centric, i.e. the distributed application components are described by using the characteristics which are related with the communication, and it hides all the other details.

4.3 The formal network model

This Section defines the entities and relationships which represents the formal network model depicted in Figure 4.1. The structure of this model is motivated by its goal which is network synthesis defined as follows.

Definition 4.1. *Network synthesis is a design process which starts from a high-level specification of a distributed system and finds an actual description of its communication infrastructure in terms of mapping of application tasks onto network nodes, their spatial displacement, the type of channels and protocols among them, and the network topology.*

Figure 4.2 reports a system under design with a partitioning between the *system portion* and the *network portion*; the former is the subject of traditional system design while the latter is the subject of network synthesis. According to the concepts described in Section 4.2 both partitions will be described with entities and relationships in this Section.

The system portion consists of *network nodes* (Section 4.3.3) which contain *tasks* (Section 4.3.1), i.e., sequences of operations accomplished to implement the overall behavior of the distributed system. *Data flows* (Section 4.3.2) are defined between tasks. Nodes are deployed in zones (Section 4.3.5), with specific environmental characteristics, and zones are related together by physical characteristics like obstacles, walls, distances,

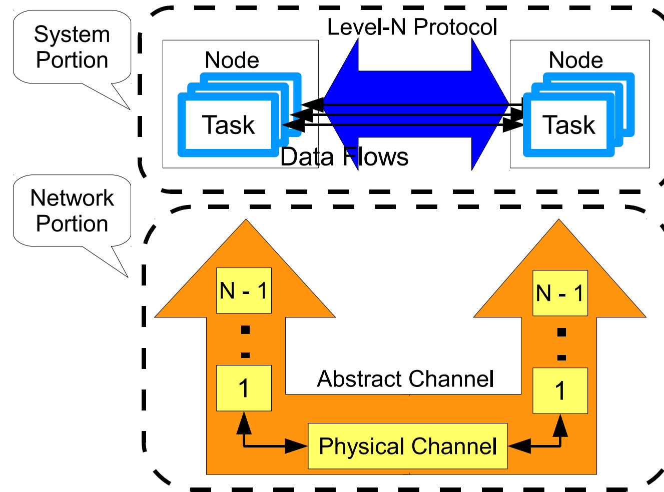


Fig. 4.2. System/network partitioning for network modeling and synthesis.

etc. (Section 4.3.6). A communication protocol is established between nodes to convey the data flows of the hosted tasks. The network portion consists of the physical channel and all the protocol entities which permit the communication between the nodes. To better describe the network portion, the entity named *abstract channel* (Section 4.3.4) will be introduced. To understand the role of this entity, let us consider two examples. In the first example, the design goal is a complete wireless device; therefore, the system portion includes CPU, memory, and all the transmission components up to the antenna while the network portion could be represented by the radio channel. In the second example, the design goal is a a temperature control application; therefore, the system portion is the control algorithm while the network portion could be a reliable byte-oriented data flow as provided by TCP/IP which hides all the lower layers down to the physical channel. The concept of Abstract Channel (AC) unifies the physical channel of the first example and the transport channel of the second one. AC is defined as follows.

Definition 4.2. *Referring to the ISO/OSI model and assuming that the functionality to be designed is at level N, the AC contains the physical channel, and all the protocol entities up to level $N-1$.*

The AC connects network nodes whose tasks are implemented by using level-N protocols. The AC is the subject of network synthesis while network nodes and tasks fall in the domain of traditional system design. According to the ISO/OSI reference model, a pair of level-N entities may communicate either directly or through a chain of intermediate systems working at lower level. In the same way, an AC may include additional intermediate systems which shall be considered both in the evaluation of its cost and during network synthesis.

In this work, all the tasks of the application under design are assumed to belong to the same ISO/OSI level, i.e., all the Abstract Channels contain the physical channel and all the protocol levels up to $N-1$.

In the rest of the Section, entities will be described in details together with relationships between them. Most of them are described as multiset, since more instances of the same type could be present in the global system design.

4.3.1 Tasks

A task represents a basic functionality of the whole application; it takes some data as input and provides some output. From the point of view of network synthesis the focus is not on the description of the functionality in itself and on its HW/SW implementation but rather on its computational and mobility requirements which affect the assignment of tasks to network nodes.

A task t is defined as follows:

$$\begin{aligned} t &= [c, m] \in \mathscr{T} \text{ where:} \\ c &\in \mathbb{R}^n \\ m &\in \mathbb{B} \end{aligned} \tag{4.1}$$

$\mathscr{T}$ is a multiset of tasks. The vector named $t.c$ represents the resource requirements to perform task's activity. For instance, in the early stage of the design flow, when detailed requirements are not available, it can be described by a single abstract number ($t.c \in \mathbb{R}$). As more details are available, it could be described by many more components, e.g., the memory usage and the computation time ($t.c \in \mathbb{R}^2$). Choosing the appropriate components of $t.c$ is a designer's responsibility. The attribute named $t.m$ is a boolean attribute representing the requirement of placing this task on a mobile node.

4.3.2 Data Flows

A data flow (DF) represents communication between two tasks; output from the source task is delivered as input for the destination task. For network synthesis, the focus is on the communication requirements between tasks which affects the choice of channels and protocols between nodes hosting the involved tasks.

A data flow f is defined as follows:

$$\begin{aligned} f &= [t_s, t_d, c] \in \mathscr{F} \text{ where:} \\ t_s &\in \mathscr{T} \\ t_d &\in \mathscr{T} \\ c &= [\text{throughput}, \text{max_delay}, \text{max_error_rate}] \in \mathbb{R}^3 \end{aligned} \tag{4.2}$$

$\mathscr{F}$ is a multiset of data flows. Source and destination tasks are represented by t_s and t_d, respectively. The vector named $f.c$ contains the communication requirements, i.e., data throughput, maximum permitted delay, maximum permitted error rate. Throughput is the amount of transmitted information in the time unit; the maximum permitted delay is the maximum time to deliver data to destination and it can be important for real-time applications; the maximum permitted error rate is the maximum number of errors tolerated by the destination. For instance, in a file transfer application no errors are permitted while in multimedia applications this requirement could be relaxed.

4.3.3 Nodes

A node can be seen as a container of tasks. At the end of the application design flow, nodes will become HW entities with CPU and network interface and tasks will be implemented either as HW components or as SW processes. From the point of view of network synthesis, the focus is on the resources made available by the node to host a number of tasks.

Formally, the node n is a tuple defined as follows:

$$\begin{aligned} n &= [t, c, k, e, p, \pi, m] \in \mathscr{N} \text{ where:} \\ t &\subseteq \mathscr{T} \\ c &\in \mathbb{R}^n \\ k &\in \mathbb{R} \\ e &\in \mathbb{R}^n \\ p &\in \mathbb{R} \\ \pi &: \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \\ m &\in \mathbb{B} \end{aligned} \tag{4.3}$$

$\mathscr{N}$ is a multiset of nodes. The multiset named $n.t$ contains the tasks associated to the node n. The vector named $n.c$ represents node's capabilities, i.e., the resources available on the node and its components have the same type as those of $t.c$. The economic cost of the node is denoted by $n.k$ and it could be estimated by referring to an actual platform. The attribute $n.e$ describes the environmental conditions supported by the node, e.g., maximum temperature, maximum humidity.

Power-related constraints and parameters can be represented by using $n.p$ and $n.\pi$. $n.p$ is the node maximum available budget, while $n.\pi$ is a function which gives the power consumption of the node as a function of the resources used by a given task and the resources available on the node itself. The latter parameter is taken into account since it could characterize the node platform. Such a function shall be specified by the designers, according with the preferred power estimation model. The total power consumption of a node can be calculated as the sum of the power consumed by each single task as follow:

$$\text{POWER} = \sum_{t \in n.t} \pi(t.c, n.c) \tag{4.4}$$

The component named $n.m$ is a boolean attribute indicating if the node can be mobile.

4.3.4 Abstract Channels

An AC interconnects two or more nodes as described at the beginning of Section 4.3.

Formally, the AC a is a tuple characterized as follows:

$$
\begin{aligned}
a &= [n, f, c, d, w, k, u] \in \mathscr{A} \text{ where:} \\
n &\subseteq \mathscr{N} \\
f &\subseteq \mathscr{F} \\
c &= [\text{max_throughput}, \text{delay}, \text{error_rate}] \in \mathbb{R}^3 \\
d &\in \mathbb{R} \\
w &\in \mathbb{B} \\
k &\in \mathbb{R} \\
u &\in \mathbb{B}
\end{aligned}
\tag{4.5}
$$

$\mathscr{A}$ is a multiset of abstract channels. The multiset $a.n$ is the multiset of nodes that communicate by using the given AC. $a.f$ is the multiset of data flows delivered by the given AC. The attribute $a.f$ cannot be omitted, since it is not always possible to get this association just by looking at tasks-to-nodes assignments. For example, if two nodes are interconnected by two or more AC's, it is required to specify throw which AC a given DF pass. The vector $a.c$ is a vector of capabilities, i.e., it represents the communication resources of the given AC, i.e., the maximum transmission throughput, the transmission delay, and the error rate. The transmission delay takes into account of the physical propagation time, of eventual packets retransmissions, and of eventual intermediate systems encompassed by the AC. It is worth noting that $a.c$ has the components of the same type of $f.c$, but the former represents the communication resources provided by the AC while the latter represents the communication requirements needed by the data flow and the involved tasks.

The attribute named $a.d$ is the maximum distance between nodes which are connected by the given abstract channel. The notion of distance is quite generic at this point and it could represent the length of a wire, the range of a wireless channel, or the maximum number of hops which can be performed by a packet. The designer has the responsibility to complete the semantic value of this attribute according to design goals.

The field named $a.w$ is a boolean attribute indicating if the AC is wireless. If at least one node binded with the AC is mobile, then the AC shall be wireless.

The economic cost is denoted by $a.k$ which can be estimated by referring to an actual platform. Since in the proposed model an AC may includes intermediate systems, their economic cost shall be considered in the evaluation of $a.k$.

Attribute $a.u$ is a boolean which holds `true` if the channel is unidirectional.

4.3.5 Zones

When designing a network infrastructure, the relationship with the environment is important. For instance, the placement of wireless access points in a building should take into account its subdivision into floors and rooms and the presence of obstacles. In another example regarding a wireless sensor network for environmental monitoring, the placement of the sensor nodes depends on the spatial behavior of the data to be monitored.

The proposed approach to capture this relationship in the formal model is based on both the partioning of the space into *zones* which contain nodes and the notion of contiguity between zones. Each zone is characterized by an environmental attribute, e.g., a temperature value in case of a monitoring application. In this way, the designer can relate the data in a given zone with the presence of nodes in that zone (e.g., at least one node per

zone is required even if more redundant architecture may be designed). The contiguity between zones is introduced to put constraints on the reachability of the corresponding nodes. In this way, the creation of network topologies can be controlled during network synthesis.

Formally, the zone z is a tuple characterized as follows:

$$\begin{aligned} z &= [n, s, e] \in \mathscr{Z} \text{ where:} \\ n &\subseteq \mathscr{N} \\ s &\in \mathbb{R}^n \\ e &\in \mathbb{R}^n \end{aligned} \tag{4.6}$$

The attribute named $z.n$ is the multiset of nodes placed in the given zone. The spatial features of the zone (e.g., its extension) and the corresponding environmental information (e.g., the value of temperature) are represented by the attributes named $z.s$ and $z.e$ respectively. The designer has the responsibility to complete the semantic value of $z.s$ and $z.e$ according to design goals. The attributes of a zone could be used to represent a set of testbenches for the application under development. For instance, in a temperature monitoring application, the inputs of the application could be the sensed temperatures. $z.e$ could be a vector of possible temperatures for a zone, and thus, each component of $z.e$ could represent a new testbench.

4.3.6 Contiguities

Zones are related by the notion of contiguity defined as follows:

Definition 4.3. *Two zones are contiguous if nodes belonging to them can communicate each other.*

Contiguity represents not only the physical distance between two zones, but it can be used also to model environmental obstacles like walls. For instance, two non contiguous zones could represent two rooms divided by a thick wall which does not allow wireless connections.

The contiguity is characterized as follows:

$$\begin{aligned} c &= [z_1, z_2, d] \in \mathscr{C} \text{ where:} \\ z_1 &\in \mathscr{Z} \\ z_2 &\in \mathscr{Z} \\ d &\in \mathbb{R} \end{aligned} \tag{4.7}$$

The components z_1 and z_2 represents the zones involved in the relationship. The attribute named $c.d$ is the distance between the zones. It must be of the same type as $a.d$ since it represents the minimum value of $a.d$ that an abstract channel shall have to connect two nodes placed in the corresponding zones. This implies that two nodes placed in the same zone are always able to communicate.

If two nodes are placed in non-contiguous zones, than they are unable to communicate directly but they shall use some intermediate systems. Therefore, the notion of contiguity is very useful to make intermediate systems explicit during the phase of network synthesis.

4.3.7 Graph representation

The entities described above can be put in relationship by using three graphs as follows:

$$\begin{aligned} FG &= (\mathscr{T}, \mathscr{F}) \\ CG &= (\mathscr{N}, \mathscr{A}, E_1, E_2) \\ ZG &= (\mathscr{Z}, \mathscr{C}) \end{aligned} \tag{4.8}$$

The Flow Graph (FG) is the directed graph in which the verteces represent tasks, and the edges represent data flows.

The Channel Graph (CG) is the directed bipartite graph, in which the nodes and the abstract channels represent verteces, while E_1 and E_2 are the sets of edges defined as follows:

$$\begin{aligned} E_1 &\subseteq \mathscr{N} \times \mathscr{A} \\ E_2 &\subseteq \mathscr{A} \times \mathscr{N} \end{aligned} \tag{4.9}$$

The representation as bipartite graph is useful to represent channels shared by more than two nodes.

The Zone Graph (ZG) is a non-directed graph in which the verteces represent the zones, and the edges represent the contiguity relationships between them.

4.3.8 Relationships between entities

The proposed entities can be used by the designer to specify the application requirements which can be expressed through formal relationships between their attributes.

Since many variables are in $\mathbb{R}^n$, mathematical operators are considered in such field. For instance, the operator $\leqslant$ induces a non-strict partially ordered lattice $\mathbb{R}^n_{\leqslant}$. Some examples of possible relations follow.

To state that the tasks associated to a node shall not require more resources than the ones available on that node, a formal constraint can be written as follows:

$$\sum_{t \in n.t} t.c \leqslant n.c \tag{4.10}$$

Similarly, a constraint on power consumption can be expressed as follows:

$$\forall n \in \mathscr{N}\, n.\pi(n.c, n.t) \leqslant n.p \tag{4.11}$$

Also spatial relationships can be easily expressed. For instance, in a home monitoring application the designer may have created several zones for each room and used the attribute $z.s$ to record the room identifier; the following Equation specifies that there must be at least one node for each room:

$$Z_i = \{z \mid z.s = i\}$$
$$\forall i \bigcup_{z \in Z_i} z.n \neq \emptyset \tag{4.12}$$

Another important design constraint could be the maximum or minimum number of nodes in each zone. This constraint can be expressed as follows:

$$\text{MIN} \leqslant \|z.n\| \leqslant \text{MAX} \tag{4.13}$$

The constraint that a node shall be mobile if it contains at least one mobile task, and that the attached abstract channels shall be wireless, can be modeled as follows:

$$\forall n \in \mathscr{N}(\bigvee_{t \in n.t} t.m) \Rightarrow n.m$$
$$\forall a \in \mathscr{A}(\bigvee_{n \in a.n} n.m) \Rightarrow a.w \tag{4.14}$$

In Equation 4.14 the operator $\bigvee$ has the meaning of the logical or.

The attribute $n.e$, which expresses the environmental conditions supported by the node, can be easily related to $z.e$, which expresses the environmental features of a zone. Formally, a node can be placed into a zone only if the following is satisfied:

$$\forall n \in z.n \;\; z.e \leqslant n.e \tag{4.15}$$

From a mathematical point of view, the ability to express relationships and equations between entities is useful to describe constraints and optimization metrics. It is worth noting that some Equations, such as 4.10 and 4.11, hold for all the systems while others depend on designer's requirements.

4.4 Network synthesis

In the previous Section all the network-related variables have been formalized. What it is left to be provided is:

- A formulation of the synthesis process (Section 4.4.1).
- A general methodology to find the solution of the synthesis problem (Section 4.4.2).
- A taxonomy of synthesis problems (Section 4.4.3).

These issues are addressed in the remaining of this Section.

4.4.1 Problem formulation

All the variables expressed in the proposed framework can be either free or bounded, and design constraints as well as optimization metrics can be expressed by using such variables. Thus, from a mathematical point of view, the formal description of design requirements can be seen as the formulation of an optimization problem.

Hence, a definition of the network synthesis problem equivalent to Definition 4.1 is:

Definition 4.4. *Given a formal description of a distributed application, a set of constraints and an optimization metric, the network synthesis is a process whose objective is to find the optimal solution, i.e., to find optimal values for the free variables.*

The optimization metric can be very complex, taking into account many parameters, e.g., power, and economic cost. Actually, the parameters have not the same importance and the designer has the responsibility to sort them according to the priority of the design goal. For instance, in a low-budget system the economic cost could be the most important optimization parameter followed by power consumption.

4.4.2 Network synthesis methodology

The optimal solution of the problem can be found analytically but this option is usually unfeasible due to the high number of parameters to be set and the search range of their values. Furthermore, it is known that similar problems are NP-hard even in specific fields as NoC [6].

For this reason, there is a lot of literature focused on alternative strategies, which can be divided into two classes:

- algorithms which can theoretically find the optimal solution provided that a long amount of time is spent;
- algorithms which use heuristics or approximations to find good solutions in a short amount of time without any guarantee to get the optimal solution.

In the first class there are techniques like Simulated Annealing while in the second class there are techniques like graph decomposition. Also simulation plays an important role in such design-space exploration since it allows to refine the analytical model thus reducing the search space.

A good starting point for future work could be the extension of the heuristics proposed for NoC's.

4.4.3 Network synthesis taxonomy

Network synthesis problems can be partitioned into four major classes.

The Optimization Oriented Problem (OOP) class represents problems where the focus is on the optimization metric. In this kind of problems, there is no interest on the quality or dependability of the solution, and the only objective is represented by the optimization metric. For instance, the building automation scenario depicted at the beginning of this chapter could involve the minimization of economical cost or power consumption.

The second class is named *Dependability Oriented Problem* (DOP). The applications in this class are characterized by the concept of quality of service or *dependability degree*, that the final solution shall meet. Typical examples can be environment monitoring applications, where tasks are usually specified, and the objective is to collect data with some degree of dependability. Other typical scenarios are remote controlling and data streaming, which involves protocols to assure a given quality of service to end users. All safety critical applications belong to this class. Since the dependability constraint shall be respected while optimizing some metrics, this class represents a more complex instance w.r.t. the OOP class.

The third class is focused on tasks or nodes deployment, and thus it is named *Scheduling Oriented Problem* (SOP). In this kind of problem, the focus is to assign tasks to nodes, or nodes to zones, in order to achieve a given goal. Distributed computing is a common example of this synthesis problem, which is well-known by people involved in NoC design [6].

The last class is the *Testbench Qualification Oriented Problem* (TQOP).This class regards problems in which the free variables of the formal model represent the inputs of the distributed application. For instance, in a temperature monitoring application, the inputs are represented by the temperature field, which can be described in the CASSE formal model by using the $z.e$ attribute of the zones. The goal of the synthesis is to find optimal values for such inputs, i.e. to create high quality testbenches, thus making testbench qualification. A common technique to find solutions for this class of problems consists of fault coverage.

4.5 Case study I: Temperature control

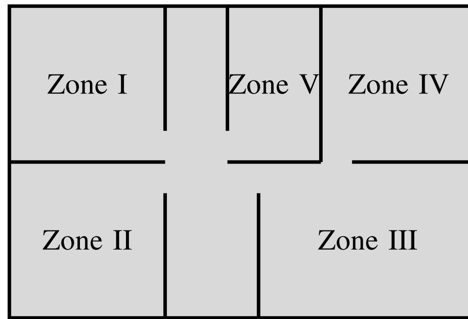


Fig. 4.3. Layout of the floor considered in the case study.

The proposed modeling methodology has been applied to a case study consisting in the temperature control application described at the beginning of this chapter and depicted in Figure 1.1. Temperature is controlled inside a building composed by floors and rooms. This problem belongs to the OOP class. For simplicity's sake, in this example a single floor has been considered, with five rooms as shown in Figure 4.3. Furthermore, to simplify the description of problem data, attribute values are reported without dimension type (e.g., bit, second, meter).

4.5.1 Tasks, data flows and zones

A set of *tasks* $\mathscr{T} = \{t_1, t_2, t_3, t_4, t_5\}$ can be extracted from a platform-independent description of the application; they are:

1. t_1: system initialization;
2. t_2: centralized temperature control;
3. t_3: user temperature control;

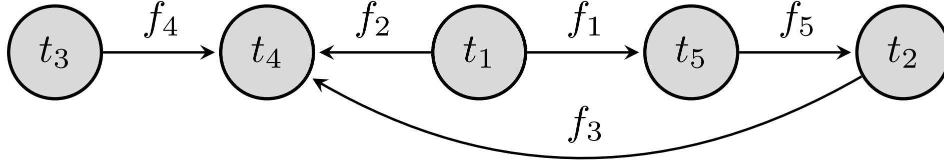


Fig. 4.4. Tasks and data flows for the temperature-monitoring application.

4. t_4: air-conditioner actuation;
5. t_5: temperature sensing.

Tasks exchange information according to the set of *data flows* $\mathscr{F} = \{f_1, f_2, f_3, f_4, f_5\}$ as shown in Figure 4.4.

Each task has an attribute $c \in \mathbb{R}^2$ which represents CPU and memory usage, and a mobility attribute $m \in \mathbb{B}$; their values are:

$$
\begin{aligned}
t_1.c &= [6,4], & a.m &= 0 \\
t_2.c &= [3,5], & b.m &= 0 \\
t_3.c &= [1,1], & c.m &= 1 \\
t_4.c &= [1,2], & d.m &= 0 \\
t_5.c &= [1,2], & e.m &= 0
\end{aligned}
$$

Similarly, each data flow has an attribute $c \in \mathbb{R}^3$ which represents communication requirements (throughput, maximum allowed delay, maximum allowed error rate); the attribute values are:

$$
\begin{aligned}
f_1 &= [t_1, t_5, [1,3,0.1]] & f_2 &= [t_1, t_4, [1,3,0.1]] \\
f_3 &= [t_2, t_4, [5,2,0.1]] & f_4 &= [t_3, t_4, [3,1,0.3]] \\
f_5 &= [t_5, t_2, [5,2,0.1]]
\end{aligned}
$$

A technological library for network nodes and channels is also available as input as reported in Table 4.1. All the types of node are listed with the corresponding capability c, coefficient vector γ (see Section 4.3.3), and price k. All the types of channels are listed with the corresponding capability c, distance d, wireless attribute w (see Section 4.3.4), and price k.

The floor in which the application will be deployed has been partitioned into a set of *zones* $\mathscr{Z} = \{z_1, z_2, z_3, z_4, z_5\}$ with the corresponding set of contiguity relationships $\mathscr{C} = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8\}$ which record the distance between zones as follows:

$$
\begin{aligned}
c_1 &= [z_1, z_2, 3] & c_2 &= [z_1, z_3, 6] \\
c_3 &= [z_1, z_5, 3] & c_4 &= [z_2, z_3, 5] \\
c_5 &= [z_2, z_5, 5] & c_6 &= [z_3, z_4, 3] \\
c_7 &= [z_3, z_5, 3] & c_8 &= [z_4, z_5, 3]
\end{aligned}
$$

	Name	c	γ	d	w	s	k	u
Node	A	$[2,4]$	$[0.8,1.1]$	—	1	—	25	—
	B	$[10,10]$	$[0.6,0.9]$	—	0	—	100	—
	C	$[2,4]$	$[1.2,0.8]$	—	1	—	80	—
	D	$[1,3]$	$[1.5,1.2]$	—	1	—	15	—
A.C.	X	$[100,0.2,0]$	—	100	0	0	5	0
	Y	$[54,2,0.1]$	—	30	1	0	20	0
	Z	$[12,1,0.3]$	—	10	1	0	10	0

Table 4.1. Technological library for nodes and abstract channels.

The distance between zones is also reported in Figure 4.5.

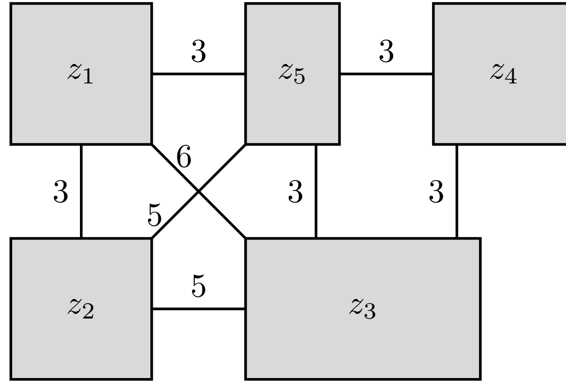


Fig. 4.5. Zone graph for the temperature-monitoring application scenario.

Finally, some constraints are given:

1. one instance of t_4 should be placed in each zone;
2. one instance of t_5 should be placed in each zone;
3. minimum five instances of t_3 must be present in the environment;
4. maximum three instances of t_3 can be present in the same zone, simultaneously;
5. maximum one instance of t_3 can be assigned to a single node;
6. an instance of t_3 is able to communicate only with other tasks in the same zone.

The goal of the design problem is to determine the number and type of nodes and the type of channels among them which minimize the total cost of the deployed application.

Considering tasks, data flows, zones and the number of deployed nodes $|\mathscr{N}|$ and abstract channels $|\mathscr{A}|$, then the total number of possible solutions would be proportional to

$$|\mathscr{N}|^{|\mathscr{T}|} \cdot |\mathscr{A}|^{|\mathscr{F}|} \cdot |\mathscr{Z}|^{|\mathscr{N}|}$$

This expression takes into account all the possible assignments of tasks to nodes, flows to channels, and nodes to zones, respectively. The number of solutions could be

very large but constraints could considerably decrease it. In the rest of the Section some simple heuristics will be used to find the solution while efficient search techniques will be investigated in future work.

4.5.2 Task assignment

Considering the nature of the given constraints the first phase of design-space exploration could be the assignment of tasks to zones.

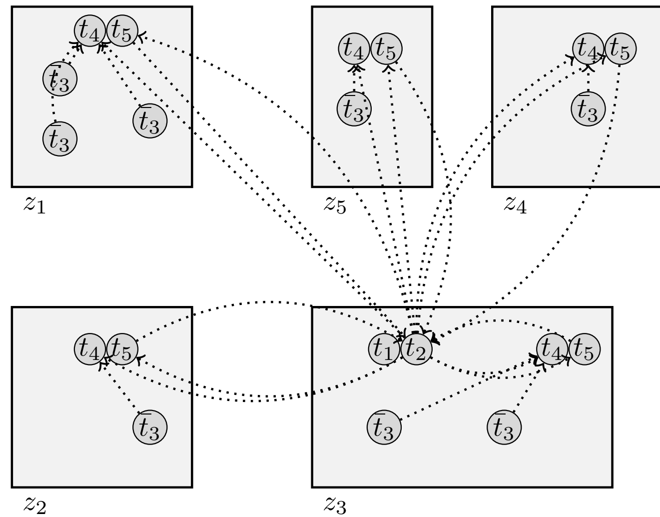


Fig. 4.6. Assignment of tasks and data flows to zones.

An instance of t_4 and an instance of t_5 are placed in each zone. Instances of t_1 and t_2 are not subjects to position constraints, so that they can be placed in any zone. The most connected zones, i.e., z_3 or z_5, can be chosen as possible candidate. It is possible to randomly choose between z_3 and z_5 (or even place one task per zone) or split the solutions' space and proceed with two or more possible solutions. For the sake of simplicity, t_1 and t_2 have been placed in z_3. Instances of t_3 are mobile (for clarity's sake in this example mobile task and node labels are denoted by a bar) so it is possible to randomly place them among all the five zones.

Figure 4.6 shows the resulting task distribution with the corresponding data flows.

4.5.3 Node assignment

Once tasks are placed, they should be assigned to nodes. Node assignment could be done by grouping tasks in the same zone into a single node (provided that node's capacity can support them). Exceptions to this method arise when a task should be mobile and others don't, or the cost of a couple of nodes is lower then the cost a single node. The process of choosing which kind of node should be picked up from the technological library could follow the rule of the *best fitting capacity* which leads to the following configuration:

$$z_1 = \{n_1[t_4, t_5], \bar{n}_6[\bar{t}_3], \bar{n}_7[\bar{t}_3], \bar{n}_8[\bar{t}_3]\}$$
$$z_2 = \{n_2[t_4, t_5], \bar{n}_9[\bar{t}_3]\}$$
$$z_3 = \{n_3[t_4, t_5], n_{14}[t_1, t_2], \bar{n}_{10}[\bar{t}_3], \bar{n}_{11}[\bar{t}_3]\}$$
$$z_4 = \{n_4[t_4, t_5], \bar{n}_{12}[\bar{t}_3]\}$$
$$z_5 = \{n_5[t_4, t_5], \bar{n}_{13}[\bar{t}_3]\}$$

where nodes $n_1, \ldots, n_5$ are implemented by type-A nodes, nodes $n_6, \ldots, n_{13}$ by type-D nodes and node n_{14} by type-B nodes.

Figure 4.7 shows the resulting assignment of nodes to zones. Table 4.2(a) summarizes tasks and nodes assignment and the choice of node types.

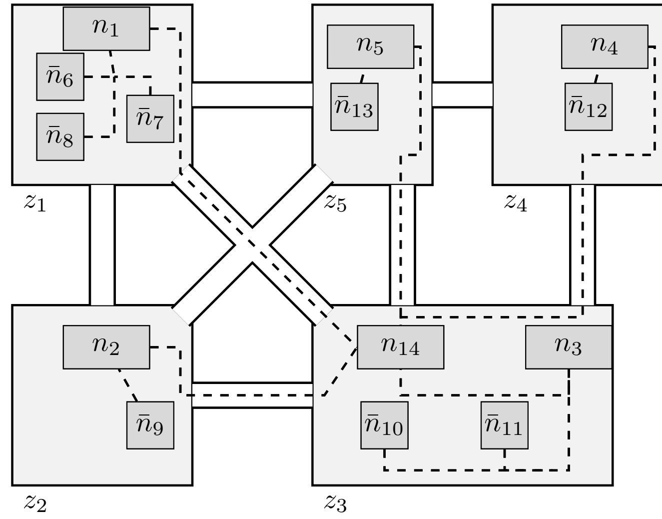


Fig. 4.7. Assignment of nodes and abstract channels to zones.

4.5.4 Assignment of abstract channels

Given the previous assignment of tasks to nodes and considering the data flows between tasks, six channels are needed to connect nodes as follows:

$$a_1 = \{n_1, \bar{n}_6, \bar{n}_7, \bar{n}_8\} \qquad a_2 = \{n_2, \bar{n}_9\}$$
$$a_3 = \{n_3, \bar{n}_{10}, \bar{n}_{11}\} \qquad a_4 = \{n_4, \bar{n}_{12}\}$$
$$a_5 = \{n_5, \bar{n}_{13}\} \qquad a_6 = \{n_{14}, n_1, n_2, n_3, n_4, n_5\}$$

In this simple case, the assignment of Data Flows to AC's is straightfoward, since each node pair is connected by at most one AC instance.

To choose the type of channels from the technological library, the following aspects must be taken into account:

- channel capabilities (in terms of maximum throughput, delay and error rate) must satisfy the requirements of data flows;
- channel distance must satisfy the contiguity between zones;
- node mobility, i.e., mobile nodes must be connected through wireless channels.

According to these issues, channels $a_1, \ldots, a_5$ could be instances of Z type which satisfies requirements concerning throughput, delay, error rate, and mobility. For instance, the maximum throughput in a_1 is 9 when the three instances of task t_3 send data to node n_1; this value is lower than the maximum allowed throughput of the abstract channel Z (i.e., 12).

The abstract channel a_6 can be an instance of X type which satisfies requirements on throughput, delay, and error rate.

The chosen abstract channels and the connected nodes are summarized in Table 4.2(b). This leads to the complete network specification, and the economic cost associated to the found solution is:

$$K = 5 \cdot A.k + 8 \cdot D.k + B.k + X.k + 5 \cdot Z.k = 400 \tag{4.16}$$

(a) Chosen nodes.

Node	Type	Tasks	Zone
n_1	A	t_4, t_5	z_1
n_2	A	t_4, t_5	z_2
n_3	A	t_4, t_5	z_3
n_4	A	t_4, t_5	z_4
n_5	A	t_4, t_5	z_5
n_6	D	t_3	z_1
n_7	D	t_3	z_1
n_8	D	t_3	z_1
n_9	D	t_3	z_2
n_{10}	D	t_3	z_3
n_{11}	D	t_3	z_3
n_{12}	D	t_3	z_4
n_{13}	D	t_3	z_5
n_{14}	B	t_1, t_2	z_3

(b) Chosen abstract channels.

A.C.	Type	Nodes
a_1	Z	n_1, n_6, n_7, n_8
a_2	Z	n_2, n_9
a_3	Z	n_3, n_{10}, n_{11}
a_4	Z	n_4, n_{12}
a_5	Z	n_5, n_{13}
a_6	X	$n_{14}, n_1, n_2,$ n_3, n_4, n_5

Table 4.2. A feasible solution for the design problem.

4.6 Case study II: Matrix multiplication

The formal model provided with CASSE is able to describe efficiently applications for distributed computation, such as matrix multiplication. This application falls in the class of scheduling-oriented problems, since the goal is the synthesis of an optimal combination of tasks and nodes to maximize computational perfomances by exploiting parallelism.

4.6.1 Network specification

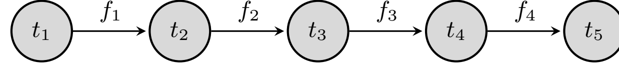


Fig. 4.8. Tasks and data flows for the matrix-multiplication application

A description of each task is provided:

1. t_1 splits input matrices into vectors;
2. t_2 splits input vectors into single values;
3. t_3 multiplies input values;
4. t_4 sums up input values;
5. t_5 builds the output matrix.

Given matrices $X_{m \times n}$ and $Y_{n \times p}$ for a time-optimal computation there will be $m \cdot p$ instances of t_2 and t_4, and $m \cdot n \cdot p$ instances of t_3.

The application for matrix multiplication is given in the form of a *flow graph* $FG = (\mathscr{T}, \mathscr{F})$, as depicted in Figure 4.8), where

$$\mathscr{T} = \{t_1, t_2, t_3, t_4, t_5\} \quad \text{and} \quad \mathscr{F} = \{f_1, f_2, f_3, f_4\}.$$

CPU and memory usage for each task are defined in $\mathbb{R}^2$ as follows:

$$t_1.c = [2,4], \qquad t_2.c = [1,3],$$
$$t_3.c = [4,1], \qquad t_4.c = [3,2],$$
$$t_5.c = [2,3].$$

Data flows have communication requirements defined in $\mathbb{R}^3$:

$$f_1 = [t_1, t_2, [5,3,0.2]], \qquad f_2 = [t_2, t_3, [3,3,0.2]],$$
$$f_3 = [t_3, t_4, [1,3,0.2]], \qquad f_4 = [t_4, t_5, [1,3,0.2]].$$

The multiset of available nodes is $\mathscr{N} = \{A, B\}$:

$$A = [t, [4,7], [50,60], \bot, 0, \bot, 12],$$
$$B = [t, [4,1], [50,60], \bot, 0, \bot, 4]$$

where $\bot$ represents a “don’t care” value for this application. The multiset of available abstract channels is $\mathscr{A} = \{X\}$, where

$$X = [n, f, [100,3,0.2], 100, \bot, 40, 0]$$

The set of zones is $\mathscr{Z} = \{z_1\}$, where $z_1 = \{n, \bot, [26,60]\}$; the set of contiguities $\mathscr{C}$ is empty.

4.6.2 Network synthesis

Since in this simple example the number of task and nodes is low, no particular heuristic have been used.

Table 4.3. Result of the synthesis process for the matrix multiplication case study.

Node Type	#	$n.t$	A.C. Type	#	$a.n$	Zone	$z.n$
A	1	t_1, t_5	X	1	A^5, B^{12}	z_1	A^5, B^{12}
A	4	t_2, t_4					
B	12	t_3					

Task instantiation

Considering matrices $X_{m\times n}$ and $Y_{n\times p}$ as input, there will be a single instance of both t_1 and t_5, $m \cdot p$ instances of t_2 and t_4, and $m \cdot n \cdot p$ instances of t_3.

Node assignment

Since the goal is optimizing the time spent by the computation, each instance of t_3 is assigned to $m \cdot n \cdot p$ instances of B (B is cheaper, $B.k < A.k$, and its capacity fits the requirements of t_3, $t_3.c \leq B.c$); each instance of t_2 and t_4 to $m \cdot p$ instances of A ($t_2.c + t_4.c \leq A.c$) and instances of t_1 and t_5 to another instance of A ($t_1.c + t_5.c \leq A.c$).

All nodes will be placed within zone z_1, since $\forall n \in \mathscr{N}\ n.e \geqslant z_1.e$ and there are no other zones.

Abstract Channel placement

Once tasks are assigned to nodes and nodes are placed within zones, data flows determine which nodes have the necessity to communicate with others. Since the available abstract channel X fits the communication requirements of all data flows ($\sum_{i=1}^{|\mathscr{F}|} f_i.c \leq X.c$), X only one instance of X is created to connect all the nodes.

4.6.3 Result of the synthesis process

The following matrices are given as application inputs:

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \end{bmatrix} \quad \text{and} \quad Y = \begin{bmatrix} y_{1,1} & y_{1,2} \\ y_{2,1} & y_{2,2} \\ y_{3,1} & y_{3,2} \end{bmatrix}$$

The flow graph with multiple task instances is shown in Figure 4.9.

The goal of this case study was the minimization of computation time. The solution given in Table 4.3 maximizes parallelism, minimizing communication delays.

Finally, the total cost of the synthesized solution is

$$\begin{aligned} K &= \sum_{n\in\mathscr{N}} n.\kappa + \sum_{a\in A} a.\kappa \\ &= 5 \cdot A.\kappa + 12 \cdot B.\kappa + X.\kappa \\ &= 148 \end{aligned}$$

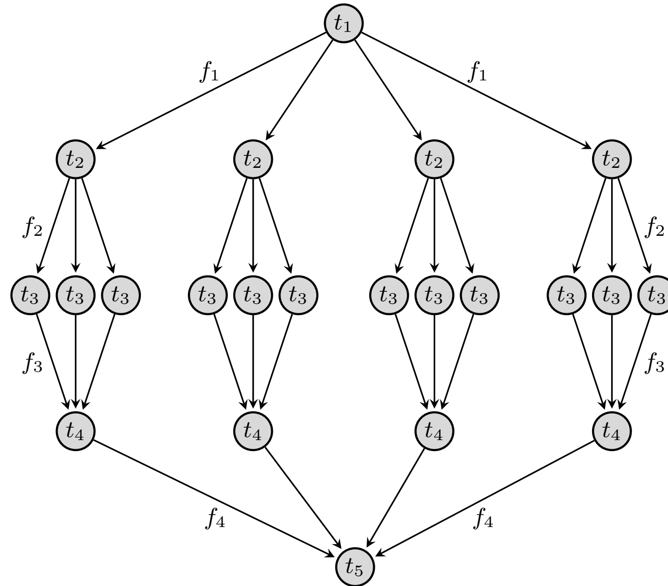


Fig. 4.9. Flow graph with multiple task instances for the matrix-multiplication application.

4.7 Conclusions and future work

An extension of the traditional design flow has been proposed for distributed embedded applications based on networked systems. Additional design phases have been introduced to model the application from a communication perspective and to provide the synthesis of the communication infrastructure. The former point is the focus of this work, i.e., the creation of a formal framework to model tasks, network nodes, data flows, channels, and interactions with the environment. Entities and relationships have been introduced through a mathematical approach which simplifies the specification of requirements and constraints. Each element has been described with examples of real-world network applications and, finally, the framework has been applied to two case studies. This work is a first contribution to a global strategy for network synthesis and thus some research topics are still open, like the possible presence of abstract channels and tasks at different ISO/OSI levels, and how to improve modeling of mobile nodes.

The work described in this chapter appears in the following publication: [75].

5

Integration of a Middleware in the CASSE methodology

The design of Networked Embedded Systems applications is a complex task due to their nature of system-of-systems in which HW nodes are connected through a network [17, 165].

This issue is faced in this chapter in the context of a middleware-based design flow.

The concept of middleware has simplified the development of embedded distributed applications by decoupling application code from the details of the HW platform and providing high-level communication mechanisms like remote procedure calls [146].

The use of middleware has suggested its modeling directly in the design flow. The Abstract Middleware Environment (AME) [69] provides an abstract model of the basic primitives of the main middleware paradigms, e.g., object-oriented paradigm. The application is written in SystemC [97] above this abstract middleware and therefore it is independent of the actual middleware, thus enhancing code reuse. AME allows the simulation of application tasks by taking into account the presence of the network but no specific methodology has been proposed to drive the mapping of tasks onto physical nodes and to define the network setup. For instance, the DSE has been performed by enumerating all possible solutions, since the example was quite trivial [69].

Network design should respect the communication requirements of application tasks which have to be extracted from application specification and represented in a formal way to be addressed in the design-space exploration. Formal representations are present in literature such as in the Communication-Aware Specification and Synthesis Environment (CASSE) [75] but no specific methodology has been proposed to extract communication requirements, nor to model middleware and to take into account during network design.

This work aims at showing that a communication-aware design flow can be created by joining together middleware programming and a communication-aware design methodology. Figure 5.1 shows the main steps of the flow; in particular, those covered by middleware programming and by a communication-aware methodology are highlighted.

The presence of a middleware layer below application tasks must be taken into account in this design flow; in particular:

1. Additional tasks (w.r.t. application tasks) are present for middleware operations.
2. Communication paths are affected by the presence of the middleware.
3. Middleware introduces a communication overhead w.r.t. to the communication requirements of the application.

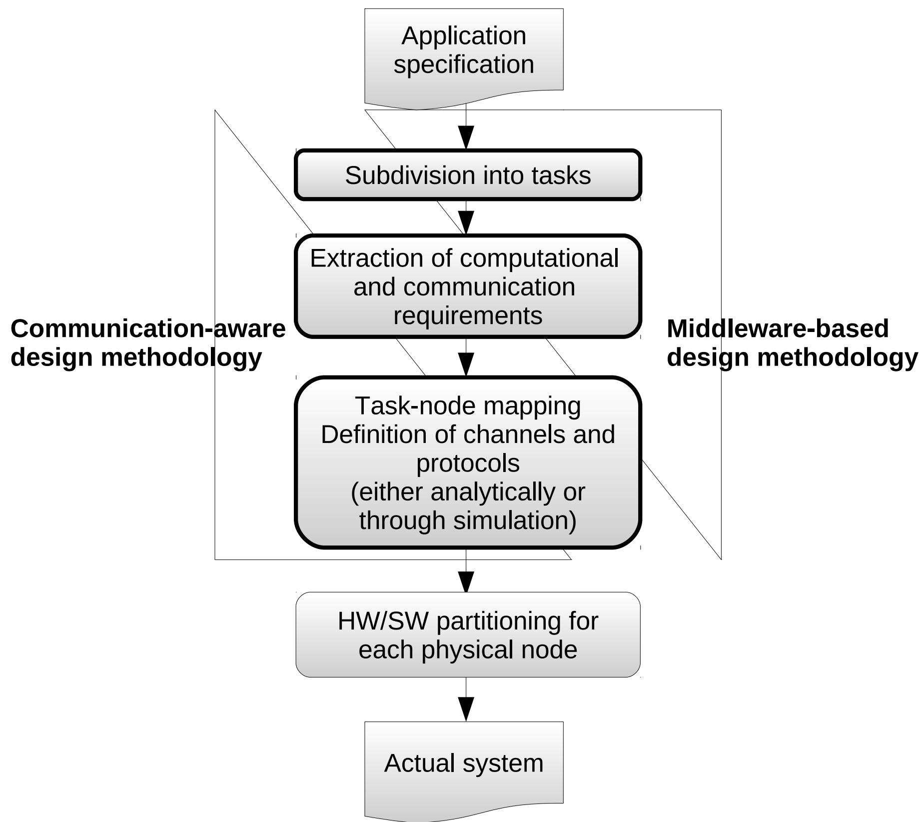


Fig. 5.1. Communication-aware design flow.

Therefore, the contributions of the chapter are:

1. A communication-aware middleware-based design flow.
2. The modeling of the middleware in the proposed framework.
3. The estimation of communication costs for a set of actual middleware implementations.

The rest of the chapter is organized as follows. Section 5.1 presents the proposed design flow. Section 5.2 addresses the presence of middleware. Finally, conclusions are drawn in Section 5.3.

No case study is reported in this section, since a complete example for the proposed general design flow which considers also the middleware as a design space dimension has been reported in Chapter 11.

5.1 Proposed Methodology

Starting from the abstract flow sketched in Figure 5.1, a more detailed flow has been derived. integrating the AME and CASSE methodologies, as depicted in Figure 5.2.

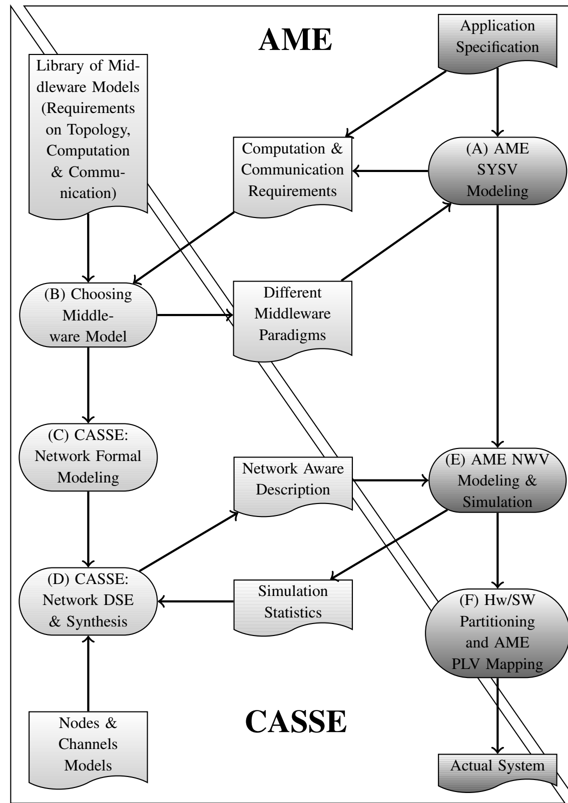


Fig. 5.2. Integration of AME and CASSE design flow.

Darker rounded boxes on the right represent the phases of the AME design flow, while the lighter rounded boxes are the phases introduced with the CASSE methodology. The squared boxes represent input/output data for the design phases.

At the beginning of the design flow, the application objectives, constraints and requirements are described at high level of abstraction and used for a first application implementation in SystemC, at AME SYSV level (Phase A).

At this point, the most important goal is the explicitation of network aspects and the contribution of AME and CASSE is relevant. AME is important since its communication primitives can be tracked in the application implementation to extract communication requirements. CASSE specification framework, on the other hand, allows to model them in a formal way. The communication/computation requirements can be either extracted directly from the application specification or estimated through simulation of the AME SYSV model (see Section 5.1.1).

In Phase B, these requirements are used to choose the most suitable middleware model inside a library in which common middleware paradigms have been classified according to a given set of relevant characteristics (see Section 5.2). Since this choice is performed at early design stage, it is possible to select a set of middleware candidates and repeat the remaining phases of the design for each candidate. During this phase, it is also possible to change the middleware paradigm. In this case, it is recommended to re-encode directly

the application in AME SYSV by using the abstract middleware with the new paradigm (re-encoding can be replaced by AME translation [70]).

After this phase, the chosen middleware and the application can be represented with the standard CASSE specification framework (Phase C). The output of this phase is a description into which middleware is modeled just as any other task. In this way, the problem has been reduced to a standard CASSE specification.

In Phase D, the formal model is solved through design-space exploration which also leads to the synthesis of the network infrastructure. Network synthesis is accomplished by choosing appropriate node and channel models.

The output of this phase is a network-aware description of the whole system in which tasks are executed inside nodes and the communication infrastructure between nodes is defined. This description can be simulated at AME NWV (Phase E) thus validating the solution. Simulation statistics can be used to drive further iterations through Phase D until application requirements are met. For this purpose, in this work AME NWV has been extended to take into account the middleware attributes chosen during Phase B. During the refinement from AME SYSV to AME NWV, the application model is unchanged. Simulative design-space exploration can be performed on the SystemC model by using automated tools which already implements many heuristics [151]. This phase shows a further contribution of AME to the CASSE design flow since it provides a simulation environment to support design-space exploration.

The remaining design phases can be carried on according to the usual AME-based design flow (Phase F).

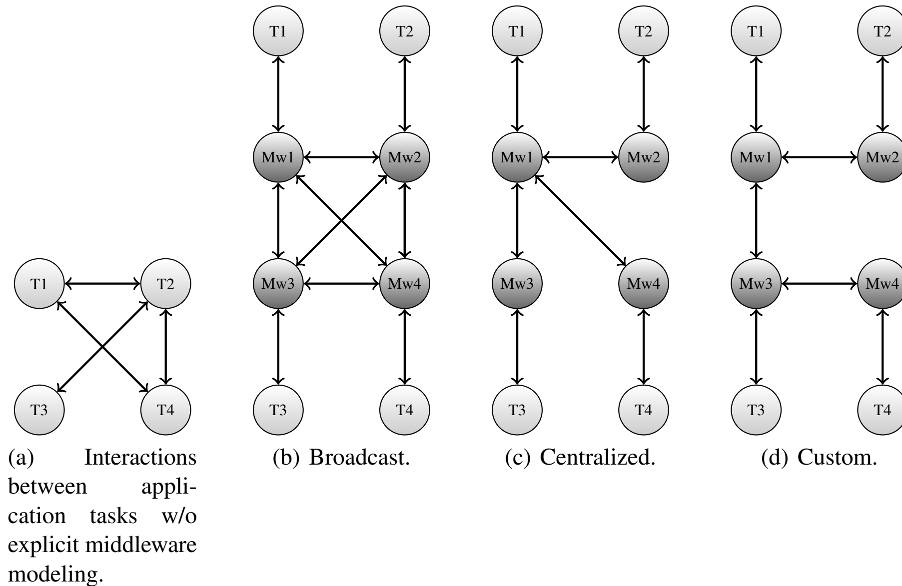
5.1.1 Extraction of Requirements from the Application

According to AME and CASSE approach, the application is decomposed into a set of interacting tasks, whose requirements need to be known, as follows:

- Computation requirements: a task, to perform the assigned operation, needs time, memory and CPU power; these requirements are used to assign tasks to nodes since each node shall be able to assure enough resources to the hosted tasks.
- Communication requirements: tasks interact each other by exchanging data; interactions can be characterized by bitrate, the maximum allowed delay and the reliability which are requirements for the design of the network infrastructure among nodes.

Such requirements could be either extracted from the initial specification of the application or derived from functional simulation at AME SYSV where a SystemC executable model of the application is present. To guarantee better results, the empirical estimation of the application requirements should be performed over a set of relevant use-cases.

Let us consider an example of the simulation approach. SystemC code can be easily annotated to track the calls to the communication primitives of the abstract middleware. Then, the generated trace can be parsed to extract the average and peak bitrate; the latter could be calculated as the average bitrate on a smaller time slice comparable to the delay constraint of the given communication.



(a) Interactions between application tasks w/o explicit middleware modeling.

(b) Broadcast.

(c) Centralized.

(d) Custom.

Fig. 5.3. Examples of Middleware Communication Schemas.

5.2 Building the Middleware Library

To integrate the AME and CASSE design flows, the presence of the middleware needs to be modeled as additional tasks and data flows as required in the CASSE specification framework. Furthermore a library of middleware models must be created starting from the set of actual middleware implementations available to the designer.

The following assumptions are given:

1. on each node, there shall be exactly one instance of the middleware task;
2. each non-middleware task cannot communicate directly with middleware tasks on different nodes;
3. middleware tasks communicate each other to create a distributed system.

The CASSE formal model allows to express easily these constraints.

From the point of view of CASSE each middleware task must be characterized according to the following aspects:

1. the communication schema followed to interact with the other middleware tasks;
2. the communication requirements;
3. the computation requirements.

The following subsections will address how to extract and represent such characteristics from an actual middleware.

5.2.1 Modeling Middleware Communication Schema

A middleware communication schema describes how middleware tasks communicates. For instance, an implementation of the object oriented middleware may contain an Object Request Broker (ORB), where remote objects are registered. Thus, each middleware

task shall be able to communicate with the ORB and with the nodes where the remote objects are deployed. However, the middleware communication schema depends only on the actual implementation of the middleware and not on its paradigm (i.e., object-oriented).

For instance, in Figure 5.3 different communication schemes are reported for an object-oriented middleware (OOM). Figure 5.3(a) represents the communication flows between application tasks before explicating middleware tasks. An OOM could implement a distributed ORB (Figure 5.3(b)). In this case, when a server task registers itself onto the ORB, just a local method call is performed. When a client will query the middleware to get the address of the server, the middleware will send the request to all the nodes (e.g., sending a packet in broadcast). The task which have registered the server will reply to the client message. Figure 5.3(c) shows a different OOM implementation in which the ORB is centralized and all registration/query messages are sent to a specific node. In this case, the node hosting the ORB will require a large amount of memory and high capacity connections. Other custom communication mechanisms could be implemented, as shown in Figure 5.3(d) in which the ORB is distributed but clients' queries follow pre-defined paths. An example in this context is TeenyLIME [35] which is a Tuple-Space Middleware which allows direct interactions only between neighbor nodes.

5.2.2 MW Computation & Communication Requirements

The requirements of middleware tasks depend on their role in the system; for example, in object-oriented middleware, the task implementing a centralized ORB requires much more memory than other middleware tasks implementing remote method invocations. The computation and communication requirements of the middleware tasks should be specified in the middleware library since, as happen for application tasks, they affect the CASSE design-space exploration, i.e., the mapping of tasks onto nodes and the definition of the network infrastructure.

Memory and CPU requirements of each middleware task can be estimated by analyzing its implementation on actual HW/SW platforms. For instance, the memory required can be estimated by checking the size of binaries and the runtime memory usage. The estimation will be more accurate if the considered HW/SW platforms are the same candidate platforms for the system to be designed.

Let us focus on the definition of communication requirements for the middleware models of the library. Usually, middleware layer does not generate data autonomously, excepts during setup, but it is driven by the communication flows of the application. For this reason, we are interested in evaluating the communication overhead as a function of the amount of application data.

Usually, middleware messages contain application data, with an abstract representation suitable to be exchanged between heterogeneous architectures, and a header, which encodes support information. For instance, in OOM, the application data are the parameters of the remote method invocation while the header could contain the destination address and the name of the method.

In this work, we assume a linear model of the communication overhead; given an amount s of application data, the actual amount of data transmitted by the middleware is o, defined as follows:

$$o = m * s + q + \text{Ack} + \text{MethodEncoding} \tag{5.1}$$

where m accounts for the effect of architecture-independent data encoding, q is the constant overhead introduced by the header, Ack is the size of a middleware ack, and MethodEncoding is the overhead introduced by the encoding of the name of the remote method.

Therefore, the next step is the estimation of such coefficients for each candidate middleware present in the library. These parameters can be estimated in two ways. The first possibility is to get them from middleware specification. This approach allows to create very detailed overhead models, but it requires a deep knowledge of middleware specification provided that it is publicly available. The other, more empirical, approach consists in tracing the communication of the actual middleware on a representative set of application scenarios. In the experimental part of this work (Section 11.3), network packets have been captured by Wireshark [79]; with reference to Equation (5.1), o has been obtained from the size of captured data by subtracting the size of network protocol headers; finally, an estimation of the coefficients has been obtained by varying s at application level.

The total number of transmitted messages has been estimated similarly, by using this formula:

$$p = a * c + s \tag{5.2}$$

where c is the total number of application remote calls, a holds 2 if the middleware sends an ack, 1 otherwise, and s represents the number of messages sent during the setup phase.

5.3 Conclusions

A communication-aware middleware-based design flows for networked embedded systems has been presented. This chapter has shown how to use a middleware and functional simulation to simplify the extraction of communication requirements, while their formal representation and network simulation can successfully support design-space exploration. Since the novel integrated approach is based on the concept of middleware, a methodology to characterize actual middleware implementations has been presented. The validation of proposed design flow is reported in Chapter 11.

The work described in this chapter appears in the following publication: [71].

6

Dependability modeling into CASSE

Distributed applications are used also in safety-critical fields, ranging from building automation, to homeland security and health care, thus requiring a dependability analysis, as additional step to the usual design methodologies [13, 90]. The analysis must focus on the correct behavior of the whole application and not on each single node. Again, the role of the network in the dependability analysis is crucial since faults may originate there or communication misbehavior can efficiently describe HW/SW faults [72]. For these reasons design and dependability analysis should be integrated in a flow which considers the network as an explicit component.

In literature, some methodologies have been proposed to perform a communication-aware NES design [15, 27, 75]. These methodologies allow the design of HW/SW blocks and of some communication aspects, but none of them integrates steps for the dependability analysis of the system.

Figure 6.1 shows the required design flow. Starting from the application specification, a network-aware NES design methodology is applied. When the design is completed, dependability assessment is performed and, if dependability requirements have not been achieved, new design constraints and guidelines are derived from the dependability analysis and used to perform a new design iteration. The requirements of this methodology are:

1. Modeling the dependability requirements of the application under design.
2. Modeling the degree of dependability of involved components, like nodes, communication protocols, and channels.
3. Assessing the dependability degree of a candidate solution.

From the best of our knowledge, none of the current NES design methodologies satisfies all these requirements. Thus, the Communication-Aware Specification and Synthesis Environment (CASSE) [75], has been taken as reference model to show how to make a communication-aware methodology suitable also for dependability analysis.

The main contributions of this work are:

1. The analysis of the system attributes relevant for the dependability assessment, with a special interest for communication-related attributes.
2. The extension of an actual NES design methodology, namely CASSE, to consider also reliability constraints for the application under design.

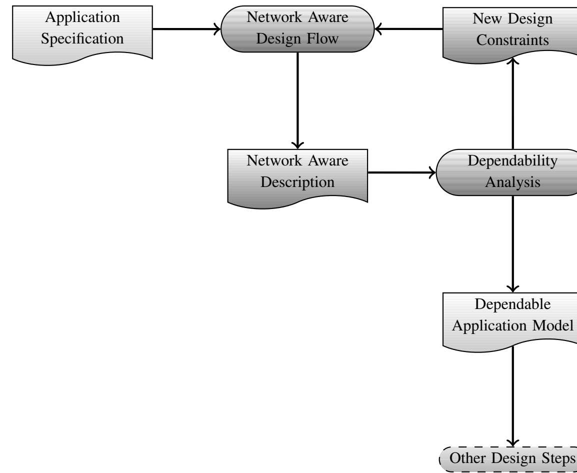


Fig. 6.1. A communication-aware design flow for dependable applications.

3. A discussion on how to assess the degree of dependability for distributed embedded applications.
4. A case study to show the application of the proposed methodology.

The chapter is organized as follows. The dependability modeling issues and the proposed methodology are detailed in Section 6.1 and applied to a case study in Section 6.2. Conclusions are drawn in Section 6.3.

6.1 Proposed Methodology

The proposed design flow is depicted in Figure 6.2. The initial steps are performed according to a communication-aware design methodology, i.e., by using CASSE. They are:

1. Modeling of the application by using the described entities. Attributes can be extracted either from application specification or functional simulation. Since the objective of this work is the design of dependable applications, the formal model has been extended to incorporate also dependability concepts.
2. Design-space exploration and synthesis of the network solutions, i.e. assignment of values to free attributes according to an optimization metric. In other words, nodes, channels, and protocols are mapped on a set of candidate models of actual components.

The output of the network synthesis is a network-aware description of the application, which fulfills all the given constraints and that optimizes some parameters (e.g., power consumption).

At this point, a dependability analysis is performed, to verify that the dependability requirements have been met. If not, new constraints and informations are extracted as a result of the analysis, and a new iteration of network synthesis is performed. These steps are repeated until all the requirements are satisfied. The output of the process is

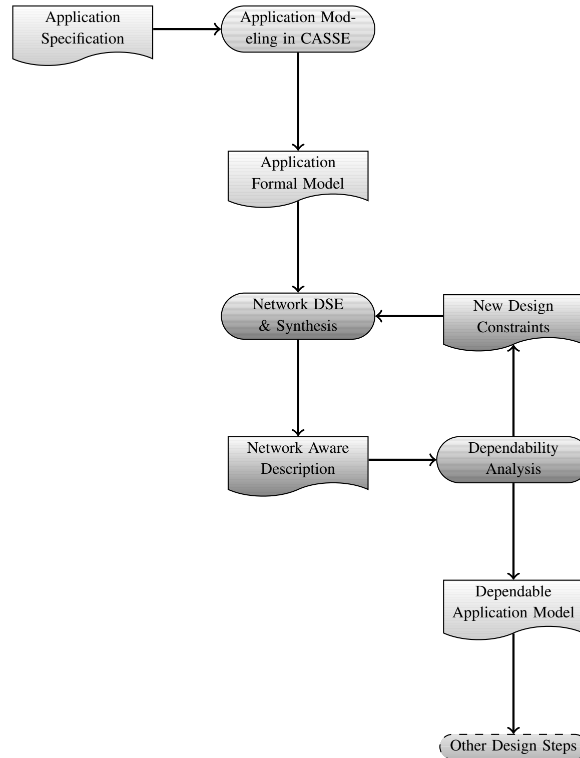


Fig. 6.2. The proposed design flow.

a network-aware system description, which also fulfills the dependability constraints. It is worth noting that DSE & synthesis must be repeated after dependability analysis to guarantee that the final dependable design solution will fulfill also the functional and performance application requirements.

Finally, other design steps can be performed for each node (e.g., HW/SW partitioning) to get the final system.

6.1.1 Modeling of Dependability Issues

The formal model must be extended to incorporate information about the reliability of the application under design. The new attributes shall be able to capture failure probability of the system, like SW bugs, HW faults, both permanent and transient, but also errors concerning communications, like packet loss and corruption, channel collisions and interferences. To model these failures, the following attributes have been introduced in the entities of CASSE:

- $t.r \in \mathbb{R}^n$: the maximum error allowed by a given task.
- $f.r \in \mathbb{R}^m$: the maximum error allowed by a given data flow. It replaces the attribute $f.c.max_error_rate$.
- $n.r \in \mathbb{R}^n$: the error probability of a given node.

- $a.r \in \mathbb{R}^m$: the error probability of a given abstract channel. This attribute replaces $a.c.error_rate$.

While $t.r$ and $f.r$ will generate constraints in the design problem, $n.r$ and $a.r$ will be computed in the solution and compared with the values provided by actual nodes and channels to find a suitable mapping for the network synthesis. The new attributes are represented as vectors to allow designers to specify complex characteristics. For instance, a node could have just one attribute, indicating the probability of hardware crash. On the other hand, an abstract channel could have two components, one for the hardware crash of communication components, and one for the error rate of the protocol stack. To allow comparisons and creation of relationships, attribute $t.r$ shall have the same dimension of $n.r$, and $f.r$ shall have the same dimension of $a.r$.

Error probabilities of nodes and channels can be compared with those of actual components which are usually estimated on a statistical basis. For example, the error probability of an actual node can be derived from its value of Mean Time Between Failures (MTBF). Thus, the various error probabilities are put in the technological library of nodes and abstract channels among which the design solution will be chosen.

6.1.2 Dependability Constraints

A formal model allows designers to express properties, relationships and constraints for the application components in a mathematical language.

Regarding the dependability, we can specify:

- **Local constraints**: regarding specific components.
- **Global constraints**: regarding the dependability of the whole system.

Usually, high-level specification generates global constraints while local constraints are related to the choice of actual components. An example of local constraint is that a task t can be deployed only on a node n with a fault probability less than the maximum error probability allowed by the task itself. This constraint can be expressed as follows:

$$\forall n \in \mathscr{N} \; \forall t \in n.t \; n.r \leqslant t.r \tag{6.1}$$

Another constraint could give an upper bound to the fault probability of the nodes as follows:

$$\forall n \in \mathscr{N} n.r \leqslant R \tag{6.2}$$

Global constraints must be linked to local constraints for the mapping onto actual components. Probability relationships can help to create this link. For instance, for an application consisting of a single acyclic data path from source to destination node, the global failure probability is derived from the global success probability, which is the product of success probabilities of traversed nodes and abstract channels as reported in Equation (3).

$$r_s = 1 - (1 - n_s.r)(1 - a_s.r) * ... * (1 - a_d.r)(1 - n_d.r) \quad (6.3)$$

If the destination receives the same data from M disjoint paths (e.g., to increase reliability) the the global failure probability is given by Equation (4) in whom p_i is the error probability of the i-th path (which can be computed recursively by using Equation (3) or (4)).

$$r_m = 1 - [1 - \prod_{i=1}^{M}(1 - (1 - p_i)(1 - a_i.r))] * (1 - n_d.r) \quad (6.4)$$

6.1.3 Network Synthesis & Dependability Analysis

The formal model captures both the functional and the dependability requirements of a distributed application, thus allowing to perform the relative design steps simultaneously. Exact analytical methods have NP-hard complexity as shown even in more specific fields like NoC's [6]. Heuristics and approximation techniques can be a feasible solution.

Another approach is simulation which is even closer to the domain of dependability analysis since fault injection and fault simulation have been traditionally applied to test the reliability of HW/SW components. In past years, many fault models have been proposed, focusing various aspects, like SW, HW, EFSM models, SystemC TLM models, and, recently, also the communication channel (namely, the Network Fault Model – NFM [72]). It is designer's responsibility to choose a fault model tailored for the application context.

6.2 Case Study

The case study shows how to apply the proposed methodology. The application consists of monitoring the average temperature of an oil plant, in order to avoid overheating. Sensor nodes transmit temperature samples to a central coordinator which calculates the average. The difference between the estimated and the actual average must be bound, even in case of nodes faults.

The first step requires application modeling by using the formal model. In this example, there are only two kinds of task, namely the sensor task t_s and the collector task t_c. There are many instances of t_s and a single instance of t_c.

Samples are represented by two-bytes integers and they are collected every 100 milliseconds, leading to a bitrate of $2bytes/100ms = 16bits/0.1s = 160b/s$ which is an attribute for the data flows between t_s instances and t_c.

The model of the physical space is divided into zones in whom the temperature is assumed constant. In this example, there are five different zones and the attribute $z.e$ contains the corresponding temperature value. Let A be the true average temperature of the whole plant. Each sensing task instance transmits the temperature value of the zone in whom the hosting node is placed. The collector task receives messages from the sensing tasks and estimates the average temperature $\hat{A}$.

Since the temperature monitoring of the oil plant is a safety critical application, there is a dependability constraint about the collected average temperature, i.e., the estimated average $\hat{A}$ shall not differ from the actual average A more than 10%, as follows:

$$|\frac{A - \hat{A}}{A}| \leqslant 0.1 \quad (6.5)$$

High accuracy in temperature estimation can be reached by lowering the probability that some zones remain un-monitored. According to Equation (4), this fact can be achieved either by lowering node error probability or by increasing redundancy. However, more reliable nodes are assumed to be more expensive and, therefore, the design process aims at finding the optimal trade-off between the total number of nodes and their reliability to keep the economic cost as lowest as possible while satisfying the constraint on temperature error.

The dependability parameters of tasks, data flows and abstract channels are set to non-influential values, since in this case we are concerning just nodes reliability:

$$\begin{aligned} &\forall t \in \mathscr{T} t.r = 0 \\ &\forall f \in \mathscr{F} f.r = 0 \\ &\forall a \in \mathscr{A} a.r = 0 \end{aligned} \quad (6.6)$$

The other input parameters are omitted, since not relevant in this example.

There are also some other constraints:

1. The channel shall be wireless, since the actual devices will be deployed in places where it is not possible to have wires:

$$\forall a \in \mathscr{A} a.w = \text{true} \quad (6.7)$$

2. Each deployed node shall be able to communicate directly with the collector node n_c:

$$\forall n \in \mathscr{N} \exists a \in \mathscr{A} n \in a.n \wedge n_c \in a.n \quad (6.8)$$

For this case study, the network synthesis is performed analytically by using a simple heuristic, i.e., nodes should be equally distributed among zones.

The synthesis output is the following:

1. There is only one instance of sensing task t_s in each zone.
2. There is only one instance of sensing task t_s in each node. Thus, there are six nodes: five for the sensing tasks, and one for the collector task.
3. There is a single abstract channel. It provides a wireless reliable protocol, e.g., IEEE 802.15.4.

The other output parameters are omitted, since not relevant in this example.

The dependability analysis is performed through fault injection. In particular, the Network Fault Model (NFM) [72] has been adopted. The desired behavior of node failure is reproduced by injecting a permanent `LOST` fault for each faulty node. The `LOST` fault drops all packets sent to the injected node. Thus, the node appears unreachable as it was damaged. The fault is injected with a rate equals to the error rate $n.r$ of the chosen node model. For simplicity's sake, the injected nodes are randomly chosen. The fault simulation is performed by using an open source network simulator, i.e., SystemC Network Simulation Library (SCNSL) [47].

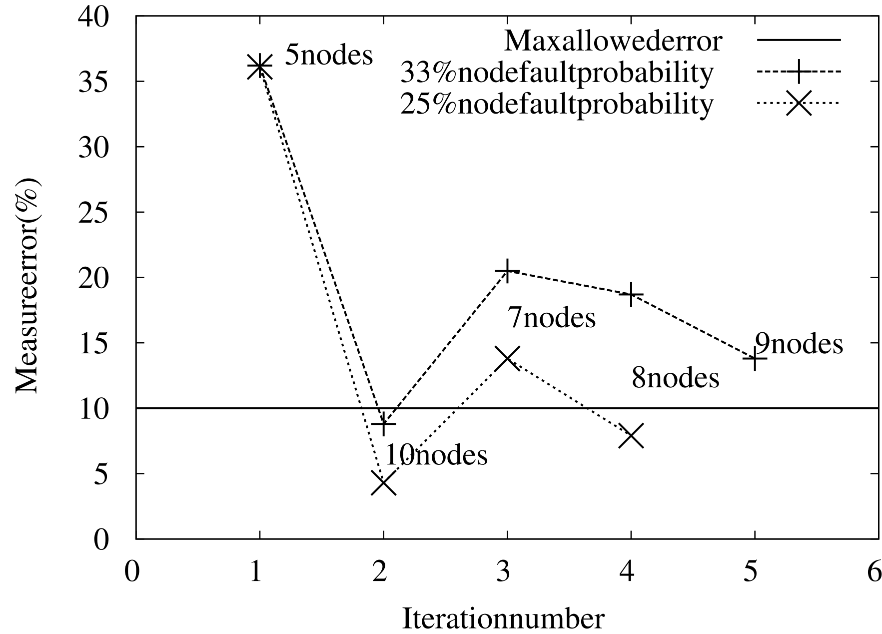


Fig. 6.3. Dependability design-space exploration for the case study.

Since the result of this first iteration shows that the dependability requirements have not been met, some constraints are created and used in a new network synthesis iteration. According to Equation (4), there are two possible constraints:

1. Use nodes with less error probability. The constraint can be modeled as:

$$\forall n \in \mathscr{N} n.r < R \tag{6.9}$$

2. Use the redundancy of nodes. The total nodes number shall not be less than a desired minimum N:

$$|\mathscr{N}| \geqslant N \tag{6.10}$$

To find the minimum-cost solution, the design-space exploration has been performed by checking different values for N and R. To minimize the number of iterations between network synthesis and dependability analysis steps, the value of N has been chosen with a bisection search strategy. Only two values of R have been considered, namely 25% and 33%. The results are reported in Figure 6.3. With a 33% error probability, the solution, achieved in five iterations, consists of ten nodes with an error of 8.8%. By choosing a more reliable node (25% of error probability), the solution, achieved in four iterations, consists of eight nodes with an error of 7.9%. The cost of these two kinds of node will allow to determine the final solution.

6.3 Conclusions

A communication-aware design methodology for dependable distributed applications has been proposed. It allows to model the dependability requirements of the application under design and the degree of dependability of involved components, like embedded devices, communication protocols, and channels. Constraints and relationships can be created to assess the dependability degree of the candidate solutions and the synthesis process can be iterated until requirements are met.

The work described in this chapter appears in the following publication: [76].

7

Network Fault Model

In distributed NES applications, the dependability must be verified not only for each single node in isolation but also by taking into account their interaction through the communication channel. Considering the example of Figure 1.1, a set of NES's interacting through the network to perform a distributed application is not so different from a set of cores in a system-on-chip or a network-on-chip. This fact suggests to consider *the whole distributed application as a single system to be verified* rather than each component node in isolation. By following this approach, inter-node interactions can be considered as those among cores in a system-on-chip. Node interactions can be very complex and dependent on the application; they range from simple point-to-point communications, e.g., to query data to sensor nodes, to many-to-many communications as in case of fully distributed algorithms, e.g., a routing protocol.

This discussion leads to the need of modeling not only HW and SW failures inside each node but also communication failures, e.g., those related to packet collisions, electromagnetic interferences and traffic congestion. Fault modeling has been addressed for a long time in different research fields such as control theory [127, 152, 167], modeling of discrete systems [38], and design of digital systems [135, 169]. Despite this past effort, in the communication field there is a lack of specific fault models. In fact, past work on the analysis of the wrong behavior of distributed systems exploited two opposite approaches:

- Studying network impact of HW and SW failures [55–57, 78, 104, 120] without considering the network itself as a source of specific failures.
- Re-creating the causes of failures in network simulations. For example, to study the effect of packet losses on a given protocol, congestions are reproduced by using interfering traffic. This approach leads to waste of time in the scenario setup phase since the modeler has to create conditions for a failure to happen.

The main contribution of this work is a novel fault model, named Network Fault Model (NFM), specifically tailored for distributed applications of embedded systems. Such a fault model has been initially thought to reproduce communication failures but it can also be used as an abstract model for HW and and SW failures which have effects on communications. The use of a network-oriented fault model requires to clarify some issues with respect to traditional fault modeling:

- The definition of fault injection.

- The points to observe the system behavior.
- The fault detection criteria.
- The temporal properties of injected faults.

This chapter describes these issues and discusses some possible solutions.

With respect to traditional fault models, the Network Fault Model has the following advantages:

- **Focus on communication**: the approach fits well with networked embedded systems.
- **Convergence**: the same fault model can be used not only to model network failures but also HW and SW failures which have an impact on communications.
- **Essentiality**: since NFM does not consider failures which have no impact on communications it can be used for an efficient first-level analysis which focuses on the most critical problems.
- **Abstraction**: the NFM can summarize many lower-level HW/SW faults, e.g., single transition faults, leading to simulation speed up.
- **Scalability**: due to the previous abstraction property, this approach works also with a high number of nodes.

Like other fault models, NFM can be used both for formal verification and for fault simulation; the latter approach is preferred when the model checker cannot verify the system in reasonable time due to its complexity and this could be the case of a distributed system made of thousands nodes. NFM and fault simulation can be used to assess:

- The dependability level of the system when it is affected by failures.
- The completeness of a testbench, by verifying if it catches all the faulted behaviors of the system.
- The completeness of the set of properties describing the system, by verifying if the corresponding checkers catch all the faulted behaviors of the system.

For the fault simulation, the NFM exploits an extension of the SystemC simulator [97], named SystemC Network Simulation Library (SCNSL) [77], which allows to model packet-based networked systems. The decision to use SystemC has the following reasons:

- **Model reusability** The model of the distributed application can be easily created by connecting together the SystemC models of each single networked embedded system.
- **Tool reusability** The same EDA tools used on the SystemC description of each single node can also be used on the whole distributed system.
- **Flexibility** SystemC allows users to describe components at different abstraction levels (e.g., RTL and TLM), by using different formalisms (e.g., finite state automata, bus-interconnected IPs, etc.), and mixing HW and SW components.

However, the adoption of SystemC is not mandatory, and the NFM can be simulated by using any other HDL.

This chapter is organized as follows. Section 7.1 introduces some background notions. Section 7.2 defines the proposed Network Fault Model. Section 7.3 describes the tools used for experimental results while Section 7.4 describes the corresponding case studies. Finally Section 7.5 draws some conclusions and remarks.

7.1 Background

In order to better understand the contribution of the work some notions are introduced here regarding fault modeling and simulation (Section 7.1.1), modeling of network interactions, and classification of the causes of wrong network interactions (Section 7.1.2). Finally, a traditional fault model is also recalled since it is used to define the proposed network fault model (Section 7.1.3).

7.1.1 Fault modeling and simulation

Fault modeling and fault simulation are two different topics. Fault modeling aims at representing wrong behaviors of the system under observation; it is based of the concept of *mutant* which changes the behavior of the system under observation. Concerning the cause of the wrong behavior, the mutants can be used to model:

- **Static errors**, that is:
  - Design errors, due to incomplete or wrong specification.
  - Implementation bugs during the development phase.
- **Dynamic errors**, which arise at operation time due to:
  - Damage of some components.
  - Malicious operations (e.g., a security attack).

Fault simulation is the use of fault models to perform different types of verification on the system under observation. In all the cases, the verification process is based on the injections of mutants in an instance of the system (named *faulty model*) and on the comparison of its simulated behavior with respect to the original system. Comparison is performed by considering a set of output of the system.

Fault simulation can be used to verify the completeness of design specification through the so-called *Property Qualification* [89]. A possible approach in this context consists in inserting blocks of code (checkers) into the model, each of them devoted to verify a part of the design specification. Then, mutants are injected into the system. Since mutants change system behavior, their presence should be detected by checkers during simulation. Therefore, undetected mutants reveal an incomplete specification of the system properties.

Fault simulation can be used to to verify the completeness of the set of the testbenches to be applied on the system, e.g., to check the presence of implementation errors. If, for a given fault the set of testbenches does not reveal a difference in the output between the original and the faulty model then such set is incomplete.

Finally, fault simulation is also related to dependability assessment since it can be used to assess the behavior of the system when some failures happen, e.g., to study its capacity to reveal the failure or to return in a safe state.

7.1.2 Representation of network interactions

A traditional formalism to represent communications and protocols is the ISO-OSI reference model [155] sketched in Figure 7.1. The communicating systems are represented as stack of computational blocks named *entities*; each entity exchanges data with a *peer entity* at the same level in the other system. Data are exchanged by following a set of rules

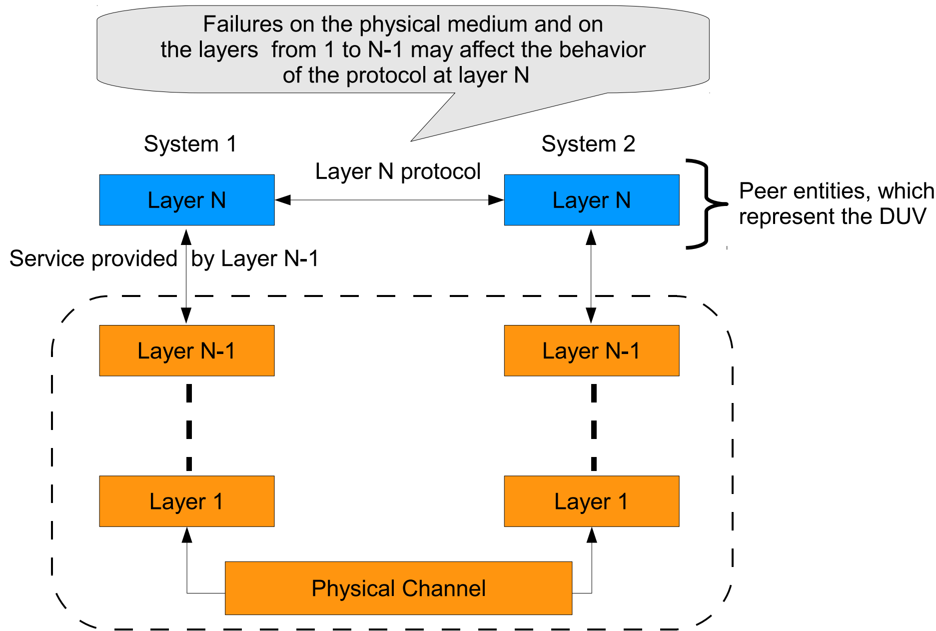


Fig. 7.1. ISO/OSI reference model with entities, protocols and interfaces.

named *protocol*. Even though, conceptually, communication takes place between peer entities, the actual data transfer is performed only between Layer-1 entities on the physical channel. In fact, each entity uses the services provided by the lower entity and messages go down through the stack to reach the physical channel. Services are defined through a standard *interface* which hides the implementation details of the lower entity, no matter if it is made up by HW, SW or other nodes. Peer entities implement a protocol by calling such services according to a defined set of rules. In literature many different formalisms are available to model a protocol:

- *Computational representation*: the protocol is described in terms of processes containing actions and conditions [84].
- *Flow chart*: the protocol is graphically represented through a sequence of steps and conditions [1].
- *Finite State Automata (FSA)*: the protocol is described in terms of a finite state automaton in which the state of transmitter, receiver and channel are represented [1, 113].
- *Petri Nets*: the protocol is described in terms of a Petri Net in which the state of transmitter, receiver and channel are represented [1].

Flow chart representation is very similar to the computational representation, and both can be easily translated into automata; also a large subset of Petri Nets can be translated into FSA. Therefore, the proposed NFM will be defined over the FSA model, without loosing general validity of the results.

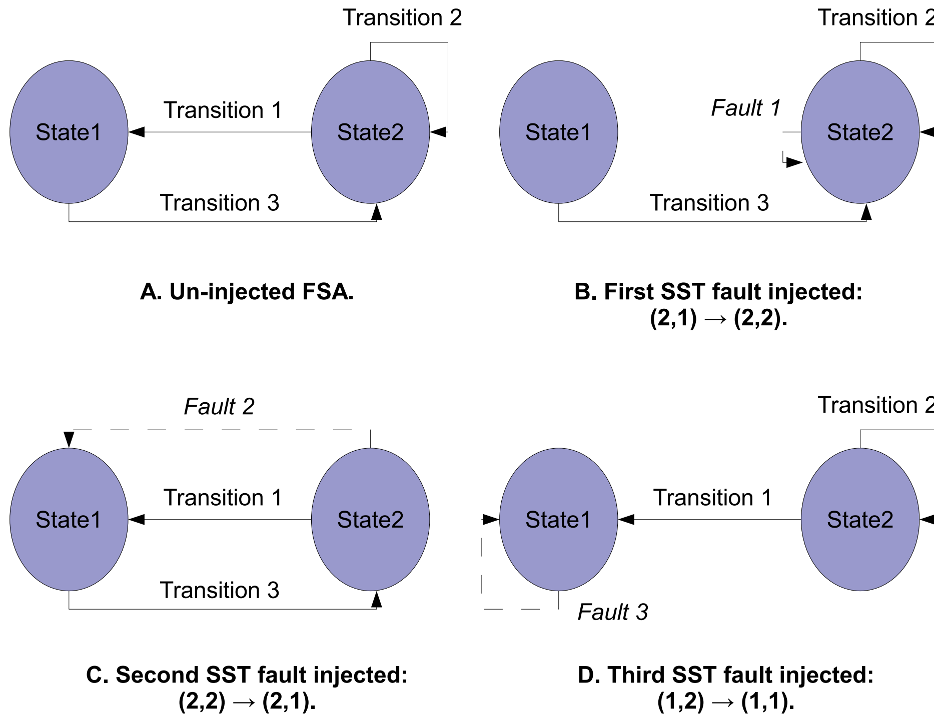


Fig. 7.2. Simple example of SST fault injection on a FSA.

7.1.3 Single State Transition Fault Model

Finite state automata (FSA) are one of the most used conceptual models to describe hardware behavior and, thus, specific fault models have been created for them, such as the Single State Transition Fault Model (SST) [38]. Figure 7.2 shows an example of finite state machine and its faulty versions. A faulty version of the automaton can be created by changing the destination of only one transition without altering its original input and output labels. Each fault is denoted as $(s, d)- > (s, d')$ where s is the source state of the transition and d, d' are the original and altered destination, respectively. Therefore, given an automaton with S states and T transitions, a total amount of $T \times (S - 1)$ faulty versions can be created.

Even though, for complex finite state automata, the number of faulty configurations could be large, SST fault model is a good abstraction for HW-oriented fault models as it allows to group many low-level failures. Experimental results show that test sequences generated with SST faults cover a very high percentage of Single Stuck-at faults and a high percentage of Transistor faults [38].

A possible extension of SST can be done by allowing to change more than one transition at one time, leading to the Multiple State Transition (MST) fault model [38]. This fault model is more complex since the different transition changes may interfere each other. Furthermore, it has been shown that sequences generated using SST, are able to detect almost all the MST faults [38].

7.2 Definition of the Network Fault Model

This section illustrates the proposed Network Fault Model. First of all, it shows how the NFM has been created (Section 7.2.1), and thus why the NFM is more abstract w.r.t. other faults models. Then, the injection policy is discussed (Section7.2.2), and a test is performed to check the NFM validity (Section 7.2.3). The option to inject the NFM as a permanent or time-varying fault is discussed. Regarding such a possibility, two issues are addressed:

- The meaning and usefulness of permanent and time-varying faults (Section 7.2.4).
- In case of time-varying faults, their activation policies (Section 7.2.5).

Finally, the need of a new observability criteria suitable for network simulations is described in Section 7.2.6.

7.2.1 FSA of the Abstract Channel

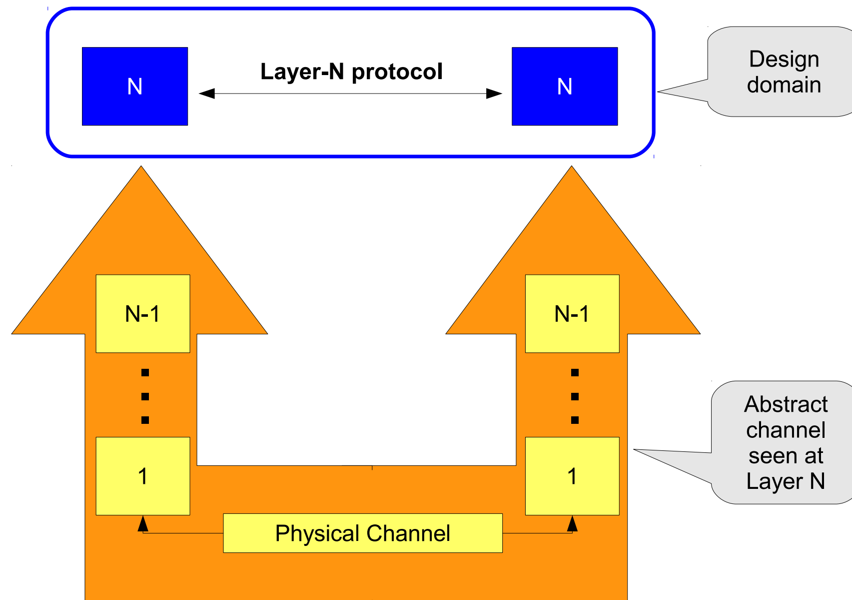


Fig. 7.3. The Abstract Channel between two peer entities.

The objective of the NFM is to represent errors occurring during communications. With reference to Figure 7.1, the focus of the NFM is the communication between peer entities, even if the reason of the fault may be in the network, as well as in HW and SW components inside the peer entities.

With reference to Figure 7.3, let layer N be the observation point of the design under verification; we define all the layers from $N-1$ through 1 and the physical channel as the *Abstract Channel* seen by layer-N peer entities. The set of faults of NFM is generated by injecting mutants in the representation of the Abstract Channel and then classifying their

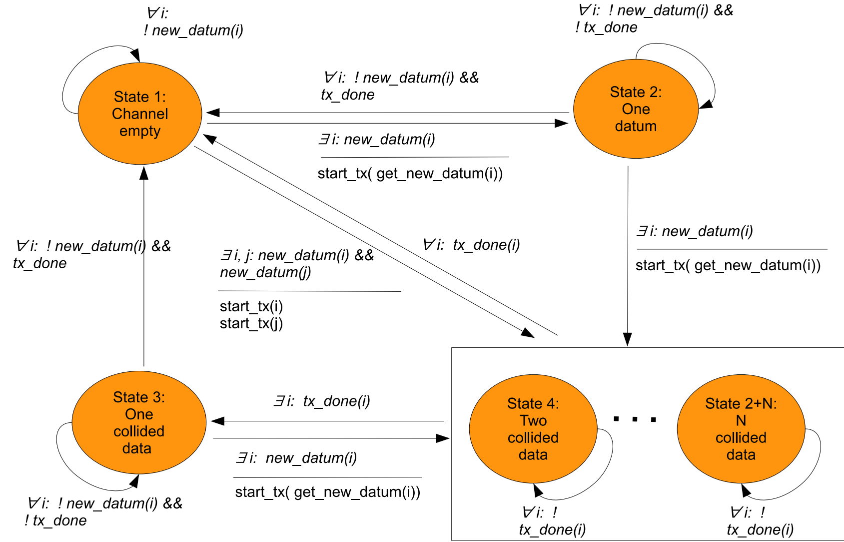


Fig. 7.4. FSA of the Abstract Channel between N peer entities.

effects. In particular, the Abstract Channel has been represented as finite state automaton (Figure 7.4) under the following assumptions:

- The FSA represents the channel state in a fixed spatial point. In general, the channel status changes continuously with respect to the space, because it also encompasses the physical transmission medium, leading to complex formal representations. In this case the channel has been simplified fixing the space variable.
- The FSA has been discretized with respect to the transmission unit, e.g., the bit or the packet. In fact, the transmission of a datum is a continuous variable. Then the FSA has been simplified discretizing the transmission events with respect to a suitable unit. Such a data unit can be a single bit for peer entities at layer one, or a packet for higher layers.
- The FSA status number increases linearly with the network nodes. The FSA shall have a status for each data collision, in order to be able to switch between the *empty*, *non-corrupted data*, and *corrupted data* behavior. Figure 7.4 represents a channel with *N peer entities*.
- Since the proposed FSA represents a channel sampled in a given spatial point, the effect of *channel memory* can be ignored. The channel memory measures the number of bits that a channel can carry without collisions, and it is defined as:

$$\text{propagationTime} * \text{channelCapacity}$$

The Single State Transition Fault Model has been applied to the resulting FSA. Table 7.1 reports the transition faults which can be obtained from the FSA of Figure 7.4, assuming $N = 2$, and classifying them with respect to their effects on the network behavior.

Five different effects can be highlighted as follows:

- **LOST**: the packet disappeared from the channel.

Network behavior	Transition faults
LOST	(1,2) → (1,1), (1,4) → (1,1), (1,4) → (1,2), (1,4) → (1,3) (2,4) → (2,1), (2,4) → (2,2), (2,4) → (2,3) (3,4) → (3,1), (3,4) → (3,3)
CORRUPT	(1,1) → (1,4), (1,1) → (1,3), (1,2) → (1,3), (1,4) → (1,3) (2,2) → (2,3), (2,4) → (2,3)
CUT	(2,2) → (2,1), (2,4) → (2,1) (3,4) → (3,1), (3,3) → (3,1) (4,3) → (4,1), (4,4) → (4,1), (4,4) → (4,3)
DUPLICATE	(1,1) → (1,2), (1,1) → (1,4), (1,1) → (1,3), (1,2) → (1,4) (2,2) → (2,4) (3,3) → (3,4) (4,3) → (4,4)
CARRIER	(1,2) → (1,1), (1,4) → (1,1) (2,1) → (2,2), (2,1) → (2,2), (2,1) → (2,3), (2,2) → (2,4) (3,1) → (3,4), (3,1) → (3,3), (3,3) → (3,4) (4,1) → (4,2), (4,1) → (4,4), (4,1) → (4,3)
Meaningless	(3,3) → (3,2), (3,1) → (3,2), (3,4) → (3,2) (4,4) → (4,2), (4,3) → (4,2)

Table 7.1. Classification of single transition faults according to their effect on network behavior.

- **CORRUPT**: some bits of the transmitted packet were flipped.
- **CUT**: some adjacent bits of the transmitted packet disappeared from the channel.
- **DUPLICATE**: the transmitted packet was sent twice.
- **CARRIER**: the test on the use of the channel is either positive or negative independently of the presence of concurrent transmissions.

The *Network Fault Model* (NFM) consists of these five network effects. Compared with traditional fault models, it provides more abstract faults which better describe network failures requiring lesser simulations. For instance, let us consider the Abstract Channel FSA; with the Single State Transition Fault Model the number of possible faults is proportional to $N * T$, where N is the number of involved peer entities, and T is the number of correct transitions. In the Abstract Channel FSA, T is proportional to N^2 and therefore the total number of possible faults is proportional to N^3. Moreover, as shown in Table 7.1, some injected faults do not represent any network behavior, they are meaningless, and simulating them would lead to waste time. Instead, many SST faults can be replaced by the same abstract NFM fault type and, at a given simulation time, at most $5 * N$ faults have to be simulated.

In conclusion, through NFM, system failure is studied from a communication perspective; communication errors are directly modeled while HW and SW errors are represented with their communication effect. It worth noting that this approach takes into account the behavior of the whole distributed system while traditional fault models are applied to each node in isolation.

7.2.2 Fault injection policy

NFM applies faults to each transmission unit, i.e., bits or packets. Considering the fault injection policy, two choices are available:

- *Injection direction*: with respect to a given node either incoming packets, or outgoing packets, or both can be affected by the NFM.
- *Injected nodes*: either a single node or all the nodes can be affected by the NFM.

Injecting incoming packets guarantees that a fault is propagated to given node, but maybe it could not propagate to other nodes. Vice versa, if faults affect outgoing packets, then they may propagate to multiple nodes, but faults could be masked due to packets collisions. In this work it has been preferred to inject incoming packets, in order to guarantee faults propagation.
When a packet or carrier fault affects all the nodes, it simulates a global channel error, like large-scale electromagnetic interference for a wireless network. Viceversa, a packet or carrier fault regarding only a single node can be used to simulate local problems, like hardware/software errors or small-scale electromagnetic interference. The first policy leads to more complex analysis, because it can be considered as a multiple-fault injection. For this paper the single-node injection policy has been adopted, to simplify results analysis.

7.2.3 Test with standard protocols

Protocol type	Fault type					Total
	CARRIER	CORRUPT	CUT	DUPLICATE	LOST	
1	5–5	0–5	0–5	0–5	0–5	5–25
2	8–6	0–5	8–8	5–5	8–8	29–32
3	10–10	0–5	10–10	1–1	10–10	31–36
4	10–10	0–10	10–10	8–8	10–10	38–48
5	10–10	0–10	10–10	10–10	10–10	40–50
6	10–10	10–10	10–10	10–10	10–10	50–50

Table 7.2. Number of detected faults as a function of transmission protocol and fault type: each cell reports the number of detected faults by considering the the secondary and the primary outputs, respectively.

To check the effectiveness of NFM, a simulation scenario has been created by using the tool described in Section 7.3. A faulty and fault-free version of a distributed system have been compared. The distributed system consists of five pairs of nodes exchanging data. Six different communication protocols taken from literature [155] have been considered as follows:

- **Protocol 1**: it is the simplest unidirectional transmission protocol since it does not care about channel reliability, buffer state, data availability.
- **Protocol 2**: it is a unidirectional stop-and-wait protocol; after sending a data packet, the transmitter waits indefinitely an acknowledge from the receiver.

- **Protocol 3**: it is a full unidirectional stop-and-wait protocol with a timed acknowledge mechanism.
- **Protocol 4**: it is a bidirectional one-bit sliding-window protocol in which data and acknowledge are sent in the same packet.
- **Protocol 5**: it is a bidirectional protocol which uses a go-back-n strategy for incorrect packets.
- **Protocol 6**: it is a bidirectional protocol which uses a selective repeat transmission for incorrect packets.

The faults are injected as permanent faults. Faulty and fault-free systems have been compared by observing two kind of outputs, i.e.:

- *Primary Outputs* (PO's): the network protocol outputs to upper layers. For instance, the outputs to the application.
- *Secondary Outputs* (SO's): the network protocol outputs to the lower layers. For instance, the outputs to the channel.

Table 7.2 reports the experimental results. Some considerations can be done:

- Since in `Protocol 1` only one node for each pair sends packets, the maximum possible value of detected faults is 5. For other faults, the maximum is 10.
- With the simple `Protocol 1`, only the CARRIER fault can be detected on SO's, since it is the only one able to affect sender behaviors. In other words, in this case the CARRIER fault represents a node failure, while other faults represent channel failures.
- Even if `Protocol 2` is very similar to `Protocol 1`, some faults can be detected also when injected on the receiver node. In fact, if something wrong happens to the acknowledge packet, the sender will stop forever to send data packets.
- Except for `Protocol 2` with the CARRIER fault, SO's have a lesser detection rate w.r.t. PO's. This make sense with the chosen injection policy, which privileges fault propagation directly to nodes. Simulation traces analysis shows that the higher detection rate of SO's w.r.t. PO's, for `Protocol 2` with CARRIER fault is caused by the masking effect of some packet collisions.
- CORRUPT fault is detected on secondary outputs only for `Protocol 6`. The protocols share the same packet format, they do not perform bit error detection, and the fault has been injected on the data field and on a non-data field used only by `Protocol 6`. This shows that the NFM helps to recognize unused fields (dead code).
- The DUPLICATE fault is more effective with complex protocols. In fact simpler protocols just ignores many packet fields: thus a duplicated packet does not force the protocol to take a different computational behavior.
- Analyzing simulation traces, it has been noted that some injected faults have been masked by packet collisions, and some applications of faulty systems have received more data than the faulty-free system. This suggests that observability criteria used for these experiments are not suitable for the NFM (this issue is addressed in Section 7.2.6).

The capacity of the NFM to change communication and application behavior, contributes to empirically support its effectiveness and its non-redundancy.

Fault Type	Injection Type	Dependability Technique	Redundant
CARRIER	permanent	host duplication	no
CARRIER	transient	transmission retry	no
CORRUPT	permanent	CRC + error correction	no
CORRUPT	transient	CRC + retransmission (ack)	no
CUT	permanent	host duplication	no
CUT	transient	retransmission (ack)	no
DUPLICATE	permanent	sequence number	no
DUPLICATE	transient	sequence number	yes
LOST	permanent	host duplication	no
LOST	transient	retransmission (ack)	no

Table 7.3. Analysis results about non-transient and transient faults.

7.2.4 The Time-varying Network Fault Model

NFM faults can be injected either as *Permanent* (PF) or *Time-Varying* (TF) faults. This feature has been introduced because the two NFM versions represent different kind of errors. In fact PFs reproduce non recoverable errors, e.g., hardware breaking or software bugs, while TFs are more suitable to represent transient conditions, e.g., channel congestions, electromagnetic interferences, and alpha particles.

An issue consists in determining if both these NFM versions make sense, and when one of them is more suitable for the system dependability analysis. A possible criterion is that TF and PF are not redundant if exists at least a dependability technique which masks only one of them.

For instance, let us consider a simple scenario in which a client node interacts with a server host in order to access some server-stored information. If LOST PF is injected, it will drop all the client requests, thus the server will be unreachable. A possible dependability technique consists in the introduction of a backup server, storing a copy of the information, and to instruct the client to contact the backup server when the main server is unreachable. Instead, in the same scenario, the LOST TF can be masked by simply adopting a confirmed protocol (i.e. an ack/retransmit strategy). Therefore LOST TF and PF can be considered non-redundant.

The case of the DUPLICATE fault is different. In fact a way to mask both DUPLICATE PF and TF is to introduce a *sequence number* inside the packet header. Hence DUPLICATE TF and PF are redundant.

Similar analysis have been done also for the other three types of faults (i.e., CARRIER, CORRUPT, and CUT). The results are reported in Table 7.3. Clearly, the masking technique for a PF is also effective for its time-varying version, because a permanent fault is a time-varying fault which is always activated, but usually the TF allows a cheaper or smarter solution. For instance, for the LOST fault, the host duplication technique is a valid solution to avoid the LOST TF, but it is more expensive and more complex than an ack/retransmission mechanism.

Analyzing the results reported in Table 7.3, the dependability techniques can be classified into two classes, according to their cost:

- *Communication cost*: the techniques are based on retransmitting lost information or adding further data inside the packets, e.g., error detecting/correcting codes and sequence number.
- *Hardware cost*: the techniques involve the duplication of a network resource, such as in case of node duplication.

7.2.5 TNFM fault activation policy

Dealing with time-varying faults leads to choose a suitable fault activation policy. There are two aspects:

- activation condition, i.e., time-driven activation vs. event-driven activation;
- deterministic activation vs. statistical activation.

The activation condition sets when and how long a fault will be activated. Time-driven activation is based on simulation time: each fault will be activated at a given instant and it will remain activated for a give time interval. In the event-driven activation policy, a fault is activated when a specific simulation event occurs. This work adopts the event-driven activation policy since the fault model is implemented into an event-driven simulator. The packet reception has been chosen as firing event.

Deterministic activation implies that when the activation condition holds, a fault is *always* activated. Instead with a statistical activation, the fault is activated with a *given probability*. This latter policy has been used for experimental results. In particular, the adopted model is able to capture:

- **The fault rate**: i.e. the average of faults occurrence in a simulation.
- **The fault burstiness**: i.e. the statistical dependency between two consecutive faults occurrences (also known as *memory effect*).

When adopting a statistical activation policy, it can be useful to test the system with different activation probabilities and activation distributions. This can allow to tune the protocol parameters and the dependability techniques to better fit the project objectives with respect to the network scenario. For instance, a noisy channel can be modeled by a statistical fault with high activation probability and/or long bursts.

To summarize, a good practice could be:

- To use a simple scenario with a random fault distribution in earlier design stages, to have a fast simulation and to quickly fix eventual bugs.
- To use the real-life scenario, with a realistic fault distribution, when design space exploration is required.

7.2.6 Observability

In the hardware field, fault simulation is based on the concept of fault propagation. A fault is considered propagated if there is a testbench which generates a difference on the *outputs*, between the faulty and the faulty-free versions of the design under verification. This common definition seems not suitable for network dependability analysis.

Selecting Secondary Outputs as observation point is incorrect. In fact if the injected fault causes a packet retransmission, then the Secondary Outputs trace will be different

with respect to the faulty-free scenario, even if at application layer there will be no difference.

Choosing Primary Outputs as observation points, raises another problem which can be explained by a simple real-life distributed scenario, made up by thousands of wireless nodes. During a non fault-injected simulation, many packets are subject to collision, while with fault injection, e.g., with the LOST fault, network performance may increase because lost packets decrease the number of collisions. This implies that it is possible that, from the communication perspective, the fault-injected scenario outperforms the faulty-free one, and hence, on the Primary Outputs there is a difference between the two systems.

This surprising result is originated by the presence of packets collisions. In fact *a collided packet can be considered a communication fault*, and hence, the faulty and faulty-free systems can be considered as systems with multiple-injected faults.

This consideration suggest that a some clarification about the concepts of inputs, outputs and propagation are required in the case of NES's. In fact, NES's are complex systems, made by hundred of different physical systems. It is designer's responsibility to define testbenches, inputs and outputs suitable for verification process.

With respect to the propagation criterion, two solutions are possible:

1. Using the hardware-borrowed fault propagation definition, applied to the outputs, but restricting the faulty-free simulation to very simple scenarios.
2. Allowing communication errors, but changing the fault propagation criterion.

Simple scenarios can be used to perform fast simulations at early stages of design, to quickly fix the main bugs and to perform simple fault propagation analysis. On the other hand, complex scenarios are slow to be simulated, and they are more suitable for advanced fault propagation analysis, when also the system development stages are almost completed.

For the complex scenarios, two new fault propagation criteria can be introduced:

1. **Quality-of-Service metrics**: these metrics are system independent, and hence can always be adopted. For example: packet loss rate and packet delay.
2. **Application metrics**: these metrics are system dependent, and hence can be used only for a specific system. For example: the speech quality in a Voice-over-IP application.

These metrics create intervals of acceptable values, which represent design constraints. A fault is then defined propagated with respect to a given metric if the faulty system *does not* satisfy constraints.

7.3 The simulation platform

This section describes a SystemC tool for NFM simulations. It consists of two components:

- SystemC Network Simulation Library (SCNSL): a library which provides network primitives required to build and simulate network scenarios.
- Network Fault Simulation Library (NFSL): a SCNSL plugin able to inject and manage NFM faults.

The SCNSL simulator is described in Chapter 10, thus it s description is here omitted. NFSL is described in the next Section.

7.3.1 Network Fault Simulation Library

The Network Fault Simulation Library has been created as an extension of SCNSL. The library main module is the `Saboteur`. `Saboteurs` are used to inject the faults, according to the values they have on their input ports. For instance, they have to know which kind of fault has to be injected and which bits to *cut* or *corrupt*. In order to be able to easily accomplish these operations, `Saboteurs` has been implemented using the `Communicator` interface.

As said above, the NFSL uses an event-driven and statistical activation policy. Hence a `Saboteur` activates itself when a packet is received by a `Node`, but before such a packet is managed by the communication protocol. When the `Saboteur` is activated, it performs the statistical check about the fault injection. The statistical check is performed through a pointer-to-function which must be supplied by user: in this way designers can adopt the probability distribution that better fits with project scenario. However, for simplicity's sake, NFSL provides five basic models:

- *Zero*: a zero probability of injection, i.e. no fault is injected. This policy is used for debugging.
- *One*: a one probability, i.e. the fault is always injected. This policy reproduces permanent faults.
- *Bernoulli* [4]: fault activations are independently identically distributed with a uniform probability density.
- *Gilbert* [4]: activation is controlled by a two-state Markov's model which allows to model fault bursts.
- *Gilbert-Elliot* [4]: this is a combination of the Gilbert and Bernoulli models in which fault activations are independently identically distributed with a uniform probability density when the Markov's model is in the fault state.

The Bernoulli, Gilbert and Gilbert-Elliot policies have been implemented directly inside the simulator because they are very common traffic models.

7.4 Case studies

Starting from the example of Figure 1.1, three different case studies have been derived, in order to show NFM potentiality. The first case study (Section 7.4.1) presents a distributed sensing application. The NFM is used to assess the dependability level as a function of the node fault ratio; this analysis aims at finding the optimal design configuration given the node fault ratio and the required level of dependability. In the second case study (Section 7.4.2), a networked the dependability level of a networked control system is assessed as a function of the communication protocol. In the last scenario (Section 7.4.3), NFM is used to obtain a significant testbench for the verification of the same networked control system described in Section 7.4.2. In all cases, the observability criterion is defined accordingly to the application.

7.4.1 Temperature monitoring

This case study aims at showing the use of NFM to design a wireless sensor network application and, in particular, to determine the minimum number of nodes required to

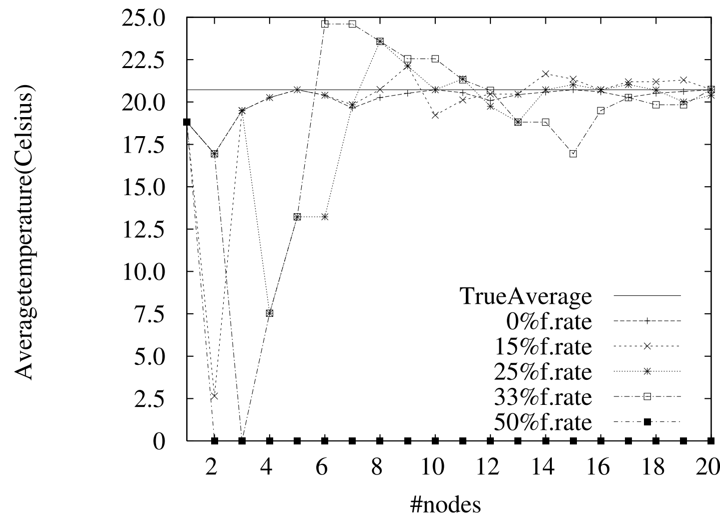


Fig. 7.5. Average value of collected data as a function of the number of sensor nodes for different values of the fault ratio.

monitor an environmental scalar value (e.g., the temperature) with a given precision and a given node reliability. A master node collects data from the deployed sensor nodes and then computes the average. The testbench reproduces the monitored area consisting of a scalar field with five sub-areas and different data values for each of them. Nodes are placed in the sub-areas with the aim of covering the whole area; nodes belonging to the same sub-area get the same sample. Network reliability is studied by using the Network Fault Model; in particular, the LOST fault type has been used. Let the *fault ratio* be the number of faulty nodes over the total number of nodes. Sampled values, nodes position and failures do not change with time. In this scenario node communications reproduce a subset of the well-known IEEE 802.15.4 standard, i.e., peer un-slotted transmissions with acknowledge [111]. For all tested scenarios the simulation length was constrained by the fault-free simulation time.

Figure 7.5 plots the average value of collected data as a function of the number of sensor nodes for different values of the fault ratio. The true value of average is 20.76 (reported as a constant line on the plot). As expected, the accuracy of the result increases with the number of nodes except with 50% fault ratio because the time wasted for retransmission to faulty nodes is too high and the longer simulation time prevents the transmission of enough data.

The reported simulation results can be used to determine the minimum number of sensor nodes required to limit the error of the resulting sampled average, which constitute the output of the NES. For example, by allowing an error of 10% at least three sensor nodes are required assuming a near 0% fault ratio while a higher number is required in case of real-world failure-prone nodes (e.g., at least nine sensor nodes in case of a 33% fault ratio). This example shows the effectiveness of the Network Fault Model to support design-space exploration.

7.4.2 Networked control system

Fig. 7.6. Architecture of a networked control system.

The proposed methodology has been applied to the design of networked control system shown in Figure 7.6; a master node sends motion commands to slave nodes which perform actions in their environment and then send feedbacks on the applied forces [126]; commands and feedbacks are delivered over a CSMA/CA packet-based network (e.g., a WLAN). This application requires a good trade-off between reliable data transmission and low delay. On one hand, packet loss, truncation and corruption may lead to inaccurate communication of commands and force feedback. On the other hand, error control leads to a high transmission delay which may compromise the control loop.

Time-varying NFM can be used to test the application in presence of failures which compromise the correct and timely delivery of data. In this way, the designer can choose the best transmission protocol among different alternatives. Thus, transmission protocols constitute the design space to be explored. The candidate protocols in this case study are:

- Protocol 1: the simplest unidirectional transmission protocol since it does not care about channel reliability, buffer state and data availability.
- Protocol 2: unidirectional stop-and-wait protocol with a timed acknowledge mechanism.
- Protocol 3: go-back-n strategy for incorrect packets.
- Protocol 4: selective re-transmission of incorrect packets.

A master-slave pair has been tested in a fault-free scenario and in presence of four types of fault, i.e., CARRIER, CORRUPT, CUT, and LOST. Fault activations are independently identically distributed with a uniform probability density function which was set to 3%. If a packet arrives later than 10 ms with respect to the previous one, it is considered lost. Each packet contains a sample of the signal exchanged between master and

Fault Type	PLR (%)	avg. IAT (ms)	dev. IAT (ms)	SLR (%)	MSE
None	0.02	1.6	0.7	0.02	0.01
	0.0	2.9	1.0	0.0	2.8E-5
	0.09	14.6	31.3	16.66	0.23
	0.0	4.4	5.0	25	0.25
CARRIER	4.0	1.6	0.7	4.0	86196.1
	0.01	4.8	13.9	1.87	0.02
	14.29	13.1	29.8	24.49	1.98
	0.03	4.9	8.8	25.02	0.25
CORRUPT	0.02	1.6	0.7	0.02	4.3E+16
	0.0	2.9	1.1	0.01	0.0
	1.67	15.2	32.5	18.55	0.64
	0.01	4.8	8.1	24.89	0.25
CUT	0.97	1.6	0.8	0.97	0.02
	0.0	4.9	14.3	1.96	0.02
	0.08	15.0	32.3	17.02	0.24
	0.01	4.9	8.9	24.89	0.25
LOST	0.97	1.6	0.8	0.97	0.02
	0.0	4.9	14.2	1.95	0.02
	0.11	15.5	32.4	17.88	0.27
	0.0	4.9	9.0	24.93	0.25

Table 7.4. Results for Protocol 1 to 4.

slave; the sample belonging to a lost packet is estimated by simply repeating the previous one (more complex concealment and interpolation techniques can be taken from the literature). Thus, the input of the NES is the transmitted control signal, and the output is the received and interpolated control signal.

As performance metrics, we considered the packet loss rate (PLR), the average value and the standard deviation of the inter-arrival time (IAT) between consecutive packets, the sample loss rate (SLR) accounting for both lost and late packets, and the mean square error (MSE) between the original signal and the received version (also taking into account the estimation of lost samples).

Table 7.4 reports the values of the five described metrics as a function of the fault type (the fault-free case is also included) for the four protocols. For all protocols, the non-null PLR of the fault-free case is due to the invalidation of network queues at the end of the simulation. As expected, the SLR is always larger or equal to the PLR and it strongly depends on delay distribution. `Protocol 1` provides a very unreliable service with high MSE even if the delay is kept low. `Protocols 2 to 4` are more robust to losses due to the use of retransmissions which, unfortunately, increase the delay. `Protocols 3 and 4` exhibit higher delay and SLR than `Protocols 1 and 2` because of the presence of a longer transmission buffer to implement the sliding-window technique.

The average number of packets generated for each of the 20 simulations (i.e., five cases multiplied by four protocols) was about 51,000 leading to a high statistical confidence. The total CPU time was about 55 minutes on the Intel Xeon 2.8 MHz with 8 GB of RAM and 2.6.23 Linux kernel (CPU time has been computed with the `time` command by summing up user and system time).

7.4.3 Testbench qualification for a networked control system

Period	Fault Type			
	CARRIER	CORRUPT	CUT	LOST
2.0	1,06E+007	1,44E+002	6,24E+006	5,64E+006
1.0	4,20E+007	4,79E+002	2,40E+007	2,27E+007
0.5	1,72E+008	5,54E+002	9,53E+007	8,97E+007
0.25	6,41E+008	5,96E+003	3,89E+008	3,73E+008
0.125	2,67E+009	7,17E+003	1,47E+009	1,49E+009

Table 7.5. MSE as a function of the testbench (sine period) and of the fault type.

The last case study shows how to use the NFM for testbench qualification. The scenario consists in the same networked control system used in Section 7.4.2. The fault rate has been set to 3% and the `Protocol 2` has been chosen for the communication. The objective of the experiments is creating a testbench which is able to stimulate the communication system effectively. As in Section 7.4.2, a valid datum shall arrive within a maximum delay of 10 ms with respect to its predecessor, otherwise an interpolation technique is applied. The MSE has been taken as observation criterion.

The control signal used as testbench is generated by using the following formula:

$$\text{sample} = 10^7 * \sin(2\pi * \text{time}/(\text{period} * \text{totalSimulationTime}))$$

Thus, different testbenches can be generated by changing the period of the sine wave. For instance, a complete sine wave is transmitted in case of `Period 1.0`, while only half wave is transmitted in case of `Period 2.0`. To achieve an high statistical confidence, during each simulation an average of 51,000 samples have been transmitted.

Table 7.5 reports the experimental results. Since the interpolation technique repeat the last received data in case of error, the MSE increases with the decreasing of the signal period, due to the increasing variance of a datum with respect to its predecessor. It is worth noting that during simulations, no data has been lost, since the protocol is confirmed, but not all samples have arrived into the maximum allowed delay. As expected, experimental results show that a sine wave with a short period can be a good testbench for this example, since it increases the MSE, whilst a sine wave with a long period is not able to affect the application QoS. In this scenario, the packets transmitted contain only computation data (i.e. sinewave samples), which have no influence on the application behavior. Thus, in the experiments, the SLR has been constant with respect to the fault type.

7.5 Conclusions

A communication-centric fault model has been proposed for networked embedded systems, able to reproduce HW/SW and network errors. Such a fault model, namely the Network Fault Model, has been derived by abstracting the faults related to the injected finite state automaton of the communication channel. All the issues regarding the adoption of the NFM have been addressed, including the permanent and the time-varying versions,

and the observability criteria. The reported case studies show the NFM effectiveness for design-space exploration, dependability assessment and testbench qualification.

The work described in this chapter appears in the following publications: [72, 74].

8

Propagation Analysis Engine

Given a design, a set of faults in the design, and a set of tests (the input stimuli), fault simulation is typically used to decide which faults can be propagated by at least one test. In complex designs, the number of injected faults is very large, thereby stressing the importance of implementing fast and efficient fault simulators. Traditionally, *serial fault simulation* is the simplest method of simulating faults, in which faults are simulated one at a time [3]. Even if this can be applied to industrial designs, it is possible to explore the potentialities of parallel simulation to further increase performances. This possibility has led to the development of special purpose computer architectures for fault simulation [20, 112, 140] at gate level. These accelerators are designed to take into account the concurrencies that exist in fault simulation. Indeed, besides serial fault simulation, there are, at least, three further types of fault simulation: parallel [149], concurrent [160] and deductive [12, 81]. These techniques differ from serial method in two fundamental aspects: (a) they determine the behavior of the design in presence of faults without explicitly changing the model of the design, (b) they are capable of simultaneously simulating a set of faults. In *parallel fault simulation*, the fault-free design and a certain number of faulty designs are simultaneously simulated. The subset of faulty designs is opportunely selected to avoid interactions among faults. *Concurrent fault simulation* is based on the observation that, generally, during a simulation session, the majority of signals/variables values in faulty designs equal the values of corresponding signals/variables in the fault-free design. Finally, the *deductive fault simulation* simulates the fault-free design to determine all of the good values on the inputs and outputs. Then, using these values, the list of all faults that cause changes in the output are "deduced" from the input values and the gate function. These techniques may also be implemented in conjunction with distributed and parallel processing [136].

When applying such techniques, fault simulation is generally performed on gate-level designs, and failures are modeled by using classical gate-level fault models (e.g. stuck-at, bridge, etc.). In particular, several algorithms have been developed, in the past, to address efficient gate-level fault simulation under the stuck-at fault model by using parallel [158], deductive [12], concurrent [109], parallel-valued list [117], differential [39], and parallel-differential [122] approaches.

Indeed, fault parallelization can be obtained also at functional (RTL) level, by running multiple instances of the design [115] on a parallel architecture-based machine. However, at RTL it is not possible to directly exploit word-level vectorization, as done at gate level.

Nevertheless, gate-level simulation is definitely slower than RTL simulation due to the additional information modeled at the gate level. Such conflicting considerations give rise to two main questions:

1. Can the *gate-level* parallelization be faster than serial *functional* simulation?
2. If yes, in which cases in which this advantage is higher?

This work tries to answer the previous questions. In particular, it discusses how to benefit from the use of *gate-level parallel simulation techniques* for simulating *functional-level faults*, along with the issues implied by porting parallel simulation from gate level to functional level. In this context, the rest of the chapter refers to the concept of simulating functional faults by exploiting parallelization techniques on a gate-level netlist with the term *parallel functional fault simulation*. The implemented parallel fault simulator has been named *Propagation Analysis Engine* (PAE).

The remaining content of this chapter is organized as follows. Section 8.1 describes the framework used to compare PAE with traditional RTL serial fault simulation. Section 8.2 details the problems arising by applying parallel fault simulation to functional faults. Section 8.3 explains some optimizations implemented to further increase parallel simulation performance. Section 8.4 reports experimental results and a comparisons between RLT simulation and gate-level parallel simulation. Finally, Section 8.5 is devoted to concluding remarks.

8.1 Framework

Figure 8.1 shows the framework that have been set up to compare serial RTL simulation with parallel gate-level simulation on the same set of functional faults. The RTL design under verification (DUV) is instrumented by injecting functional faults and a set of testcases are generated by exploiting a functional Automatic Test Pattern Generator (ATPG) proposed by the authors in previous works [50]. Then, the instrumented DUV is synthesized to obtain a gate-level netlist. Finally, the testcases are simulated on both the RTL faulty designs, by exploiting a serial simulation engine, and the gate-level faulty netlist, by exploiting a parallel simulation engine. The fault propagation results is definitely the same, but the simulation time may sensibly vary as reported in the experimental results section.

8.2 Open issues

The parallel functional fault simulation is a complex task with many tradeoffs between design decisions. The main and most important ones are the followings:

- the adopted fault model, which shall be behavioral but synthesizeable;
- the choice of which kind of functional fault parallelization shall be used;
- the parallel faults management engine and its integration with the parallel netlist;
- the simulation kernel and the adopted simulation language;

Each of these problems is addressed in the next sections.

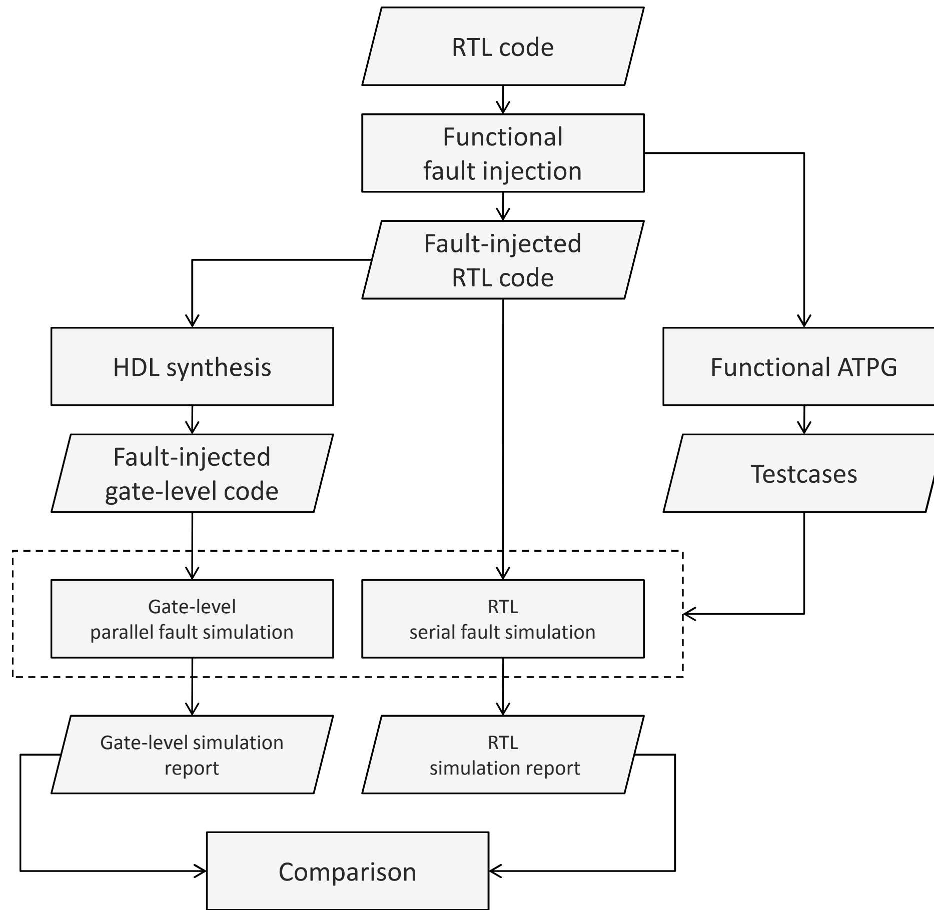


Fig. 8.1. Framework.

8.2.1 The functional fault model

In the literature there is a great variety of different fault models, and each of them has been developed to focus on particular design problems and design levels [2, 83]. When approaching the parallel simulation at functional level, not all the fault models are suitable. In fact the parallel simulation is implemented at gate level, and hence, the injected design must be synthesizeable. This implies that the chosen fault model shall not introduce non-synthesizable constructs. Moreover, it is required that the injected designs have a new port (or variable), namely the *fault port*, which is used to select which fault to enable, because, after parallelization, this port shall be driven and managed by the parallel simulation engine (Figure 8.2).

For our experiments the *mutant fault model* and the *bit coverage fault model* have been chosen because they meet this requirement.
The bit coverage fault model [61] has been developed for the RT level, with the objective to correlate the RTL faults with the gate-level ones [60, 67]. This fault model is very

similar to the traditional gate-level stuck-at fault model, because it sticks a bit either to zero or to one as stuck-at does. Moreover, it can also stick a condition to *true* or *false*. One of the main differences between these two fault models is that bit coverage is at the functional level, and it is designed to inject faults into variables, functions and operations, rather than gate-level nets. Hence, it has to perform complex injection operations because it has to deal correctly with all the RTL data types.
The mutant fault model is far more abstract than the bit coverage, and it is more focused on functional verification [48]. In the past testing based on mutation has been depicted as powerful but computationally expensive. This expense has prevented mutation from becoming widely used in practical situations, but recent engineering advances have provided techniques and algorithms for significantly reducing the cost of mutation testing [129, 130]. There are a lot of different kinds of possible code mutations: a statement can be deleted, a condition can be stucked at *true*, *false* or it can be negated, an arithmetic operator can be substituted with another one, and each boolean relation can be substituted with another one.
In our experiments, the usage of these two different fault models, which are also related with different verification objectives, points out that the explained techniques can be widely adopted, even in very different context.

8.2.2 Functional fault parallelization

Among all possible parallelization techniques, *vectorization* and *concurrency* have been used. The *deductive* technique has not been considered, because it requires to modify the simulation kernel. On the contrary, vectorization and concurrency can be implemented with a hardware design language and then simulated by using the designers preferred simulator.
Vectorization can be applied only to bit-blasted netlists. Bit-blasting is the term for breaking down each netlist element to its individual bit members. The vectorization is implemented by mapping each single bit to a vector of bits. Then each operation on bits is transformed into an operation on such vectors. It is done in order to associate each bit of the vector to a netlist copy, enabling the faults on all the copies but one, namely the "reference copy", which performs the normal computation. Inside the netlist it is required to transform each computation on bits to a computation on bit-vectors, i.e., to transform a logic operator to the corresponding bitwise one. This task is quite simple because all the usual languages support bitwise operations natively. Figure 8.2 shows the steps required to implement this parallelism. In this implementation, which is written in *C*, each vectorized bit has been mapped on a machine word, in order to use directly the machine instructions, avoiding the possible overhead of using more complex constructs. This mapping to a machine word allows switching from a 32-bit machine machine to a 64-bit one, to further increase performance, without any code changing.
The other kind of parallelization implemented is *concurrency*. It consist in enabling more than one fault per netlist copy. The main problem is that if two faults will impact on the same outputs or on the same registers, *a priori* they cannot be classified because it is not possible to understand their interaction and their effects in the simulation. To avoid this problem there are two possible strategies. The first strategy is to enable different faults as long as they will not interact, i.e., their impacted registers and outputs will never interact. This check can be performed offline, but it can require some long computation time.

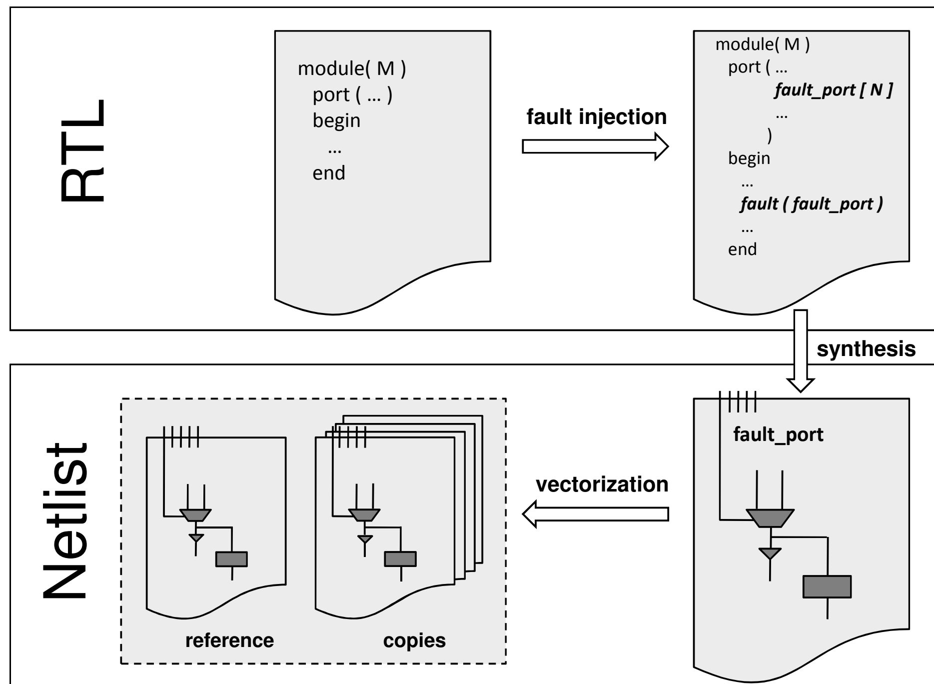


Fig. 8.2. Steps required to vectorize functional faults.

Moreover in real-life netlists, each fault impacts on many registers, and hence this implies that only few faults could be enabled at the same time. The other solution is to just enable the faults without any offline check, but the netlist has to be instrumented in order to check eventual conflicts at runtime, i.e., during the simulation itself. This means that if a fault is going to conflict, it must be disabled before this will happen, and all the impacted registers and outputs must be reset to the reference netlist value. Moreover as soon as a fault is classified as propagated, it can also be disabled, in order to minimize the conflicts. In PAE this second strategy has been implemented.
In the rest of this chapter, where not specified differently, with the term "parallelism" refers to these two joint parallelisms.

8.2.3 The parallel simulation engine

While the injected netlist must be synthesizable, this is not the case for the engine which checks the faults propagation, and manages the parallel faults simulation. It could be useful to synthesize also the management code in order to deploy it on an hardware accelerator [31], but due to its complexity and to the high cost of hardware accelerators, this idea could be addressed in some future work. For this reason, in the developed parallel simulation engine there are two main conceptual modules: the parallelized netlist, and a wrapper. The problem is that it is not easy to integrate these parts, due to their different abstraction levels:

- The wrapper has to read the inputs to be passed to the netlist, but such inputs are designed for the RT level. Hence, the wrapper has to split them into single bits, and then to vectorize them, in order to be applicable to the bit-blasted and vectorized ports of the netlist.
- The outputs of the netlist are bit-blasted and vectorized. Hence, the wrapper has to compare each bit to check differences between the reference netlist and the injected netlist copies. A naive approach could considerably increment the simulation time: hence, it is required to implement an optimized algorithm which uses bitwise operations and low-level bit manipulation functions.
- The registers must be taken into account for fault concurrency on the same netlist copy. In fact, each low-level register has been substituted with a function which implements the register itself but also checks for fault collisions. Moreover, when a collision happens, one of the colliding faults must be disabled, restoring its mutated registers to the reference copy value.
- The fault port must be vectorized and split as each other netlist input. Moreover a mechanism must be implemented to enable multiple faults on the same netlist copy. A lot of fault injection mechanisms use an integer for the fault port type, in order to identify which fault to inject. For our purposes it is better to use a fault port with a bit for each fault, and hence its vectorization and management is as simple as every other input port.

8.2.4 The simulation kernel and the simulation language

Usual hardware description languages (HDLs), like VHDL, Verilog and SystemC, are event driven. Their advantages are simplicity, and the existence of a lot of tools that have been developed to deal efficiently with them. But at gate level, the number of events is substantially higher with respect to RTL, especially with a bit-blasted netlist. This implies that performance is degraded because most of the simulation time is used in managing and dispatching the events, instead of simulating the design. To avoid this problem, it is necessary to switch to a simulation schema that is not event-driven but cycle-based. In fact cycle-based simulation is targeted to deal with few (or no) events, but it increases the computation complexity. In other words, each event must be translated into an equivalent expression, and such expressions must be evaluated following a special order, which must lead to the same computation results as the event driven model. This is not only a choice related with the simulation engine performance, but also a choice about which language to adopt: on one hand, HDLs are event-driven but they are widely used by designers; on the other hand, a language like C guarantees high computational performances, but it is not suitable for a large part of designs. The proposed solution exploits a translator, which has as input a standard HDL netlist, and it automatically generates a netlist written in C, which uses a cycle-based-like technique. The generated C netlist has to follow the schema depicted in Figure 8.3, in order to have the correct simulation results. The main point of this hybrid schema is that delta cycles are emulated by cycling through the netlist, but they are performed only when there is a new register value to be propagated. This optimizes the iterations, grouping all the events together. In fact the logic is computed following a top-down order, i.e., the output of a gate is computed before computing other gates impacted by this output: this reordering removes the need for the events, and hence all the values are directly propagated, instead of requiring a delta-cycle, like happens, for instance, for

signals in usual netlists. Instead, the registers preserve the need of two phases, one of computation and one of propagation of the new values: this is why the writing on their output nets is performed *at once*. The outputs are checked only at the end of delta cycles, in order to avoid to check values during eventual glitching.

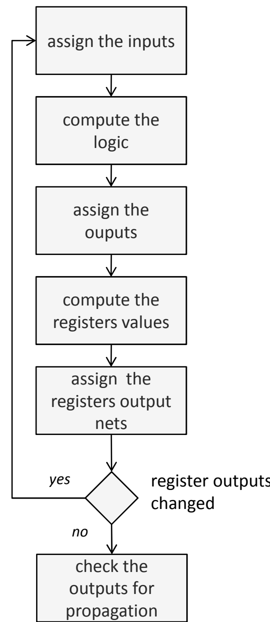


Fig. 8.3. The simulation algorithm.

8.3 Optimizations

Having resolved open issues as reported in the previous section, and implemented the basic parallel simulation engine, it is possible to investigate some different ideas to further optimize the performance:

- optimizing the inputs management;
- optimizing the mux computations;
- splitting the netlist in logic cones;
- optimizing the flops computations;
- dealing with the compilers;
- adopting a four-values logic;
- exploiting the function inlining.

All these optimizations have been implemented in our parallel simulation engine, and each of them is detailed in the following sections.

8.3.1 Optimized inputs management

The wrapper must split and vectorize all the input values of the netlist. This operation is quite slow, thus a good idea consists of checking if an input is changed, and in this case to recalculate only the associated input port values.

8.3.2 Mux computation optimization

Each mux can be seen as a sort of conditional construct. Then the mux can be rewritten as an *if* construct, into which the *guard* will be the selector, the *then* branch will normally compute the mux result, while the *else* branch will just propagate the input to be propagated in the case of the selector holds zero. The idea is that, if all the vectorized bits of the selector are at zero, then there is no need of computation. Moreover. it is possible also to encompass inside the *else* branch all the logic which impacts only on the zero-propagated mux input. In this way, it is possible to avoid many computations.

8.3.3 Splitting the netlist logic cones

Recomputing all the logic expressions at each simulation iteration is very time consuming. But usually only some expressions must be recomputed, because large parts of the netlist do not change their values. Hence, it is possible to divide the logic into different groups, which are classified with respect to the inputs and outputs of the logic. The main groups are:

- Logic driven only by inputs: this logic must be updated only when an input changes. During the delta-cycles this group will never be recomputed, because the inputs will preserve their values.
- Register driven logic: this logic is driven only by registers. Hence, it must be computed only during delta-cycles.
- Data logic: this logic impacts only the data input signals of flip-flops. Then it must be computed only when there is a “rising edge” on the clock.
- Mixed logic: this is all the logic which is not inside any other set. This logic must be always recomputed, and hence its computation cannot be optimized.

Each split part is written encompassed by an appropriate conditional construct, so it shall be recomputed only when it will be really required.

8.3.4 Optimizing the flops computations

The flop values are updated only on a clock “rising edge”. But usually large groups of flops are driven by the same clock signal, thus it is useless to perform a check for each of them. The idea is to encompass all of them inside a unique conditional construct, in order to minimize the checks. This implies that, if the flop has also asynchronous inputs, as it can be a reset signal, the flop implementation must be divided into both the conditional branches: the code to manage the synchronous signals be put inside the *then* branch only, while the asynchronous signals will be managed inside both the branches.

8.3.5 Dealing with the compiler

When dealing with low level design representations, like netlists, HDL compilers have to deal with the following problems:

- manage files with thousands of instructions and declarations inside the same scope;
- compile a large number of files in a short time.

Usually, for traditional HDLs this is not a problem, because they are languages with simple semantics and constructs.

However our parallel simulation engine has been prototyped in SystemC, which is a C++ library, and which is usually compiled by software-oriented compilers, such as *GNU gcc*. To avoid the arising compilation problems, these steps have been performed:

1. Eliminate HDL code to avoid mixed C++/HDL language compiler and simulator, as, for instance, Modelsim. This speeds up the compile time.
2. Break the netlist implementation into many short functions. In fact, it is a software engineering error to implement C++ functions made by thousands of lines, because it ends up with unmaintainable code. For this reason, *gcc* is not able to compile such huge functions. Breaking the netlist in small functions avoids this *gcc* limitation.
3. Declare all the netlist variables as globals. This avoids the scope problems which could arise when splitting the netlist implementation into many functions.
4. Implement the parallel simulation engine in C. Note that, C is a very simple language in comparison with C++. In fact, C++ compilers have to deal with complex classes, and complex resolution rules. Switching to C code sensibly reduces the compiling time.

These optimizations reduced the compilation time by three orders of magnitude.

8.3.6 The four value logic

One of the most used HDL data types is the *logic* type, which is able to describe complex bit behaviors. At gate level, all the components are usually described by using such a type, and all gates and register behaviors are described taking into account all the possible logic values. As explained before, the parallel simulation engine has been implemented by using the *C* language, to optimize computation performances, but such a language does not natively support logic types. To obtain the same behavior of a netlist described in an HDL, a logic with four values has been adopted, i.e., *zero*, *one* , *unknown* and *high impedance*. each value has been described by using a couple of bits, and then ll the logic gates have been encoded by using *Karneaugh maps*. This implementation is highly optimized, because it uses only few bitwise operations and it is faster than other strategies, like, for instance, the *lookup tables* used by SystemC.

8.3.7 Function inlining

A widely implemented software optimization is the inlining of functions. In *C* there are at least two ways to implement this optimization:

- implement the function as a preprocessor macro;
- using the keyword *inline*.

Using a macro guarantees that the function body will be inlined into the final code, but on the other hand, the final executable could suffer from "code bloating", i.e., it could be unnecessarily huge, causing a performances loss. This usually happens when the inlined function is very long. Instead using the appropriate keyword is safer, because it is a sort of hint for the compiler: in fact it is up to the compiler to decide whether to inline the function. The drawback is that the compiler may not inline the function, whereas the programmer knows that it really leads to a good optimization.
In our parallel engine there are a lot of small functions, each of them implementing a different kind of logic gate. Such functions are implemented in few lines of code, because they are highly compact and optimized using bitwise operators. Hence, such functions have been re-implemented as macro-function, in order to avoid the overhead of their calls.

8.4 Experimental results

When a simulation is going to be performed, one of the factors which has major impact on the performances is the abstraction level at which the design is described. In fact, the simulation at RTL is much faster than the simulation at gate level, because, with less abstraction, a lot of new details must be taken into account during the simulation. In our case, there is no interest in a pure simulation at gate level, but the objective is to check the faults propagation. In other words, even if the single run at gate level is slower than a single run at RTL, maybe the overall faults classification time is lower, thanks to the parallel approach that is not applicable at RT level.
To check if this is possible, some designs have been injected with both bit coverage and mutant fault models. The designs are a 12-bit microprogram sequencer, named *am2910* [5], a CPU core, named *hc11* [85], and few tests taken from the ITC-99 benchmark suite [141]. For each design, 60 input sequences, each one composed of 100 test vectors, have been generated by using a genetic engine-based functional ATPG. The faults have been classified both at RTL, by using a serial simulation engine (SSE), and at gate-level, by using the parallel simulation engine (PSE). Execution has been performed on an eight-processor Intel Xeon 2.8 MHz equipped with 8 GB of RAM and 2.6.23 Linux kernel. CPU time has been computed with the `time` command by summing up *User* and *System* time, and it is expressed in seconds. Experimental results are reported in table 8.1.

Design	**Fault model**	**Faults Number**	**SSE time**	**PSE time**	**Fault coverage %**	**Speedup %**
b10	bitcov	244	9.647	7.504	91.0	22.21
b04	bitcov	398	2.816	14.194	99.0	-403.87
b10	mutant	185	25.319	7.984	81.1	68.46
b04	mutant	66	10.337	6.728	74.2	34.94
hc11	mutant	2245	811.115	204.341	49.8	74.81
am2910	bitcov	3608	755.82	505.23	83.3	33.15
am2910	mutant	2763	2219.46	636.98	76.5	71.3

Table 8.1. Experimental results: comparison between serial and parallel simulation.

Columns report the name of the DUV (*Design*), the considered functional fault model (*Fault model*), the number of injected faults (*Faults number*), the simulation time achieved by using the RTL serial simulator (*SSE time*), the simulation time achieved by using the gate-level parallel simulator (*PSE time*), the achieved fault coverage (*Fault coverage %*), and the speed up gained by using the PSE simulator instead of the SSE simulator (*Speed up %*).

It is possible to note that for the majority of the designs, using the parallel fault simulation gains a good speedup. On the *b04* design, using the bit coverage fault model, the parallel simulation is very slow with respect to RTL. The cause of this behavior is that almost all the faults have been propagated by the first testcase. This means that the parallelism has not been fully exploited, because there were too few faults to be simulated in simulation runs after the first.
Hence, in order to optimize the overall performances, it could be a good idea to use the parallelism at the beginning, and when the faults to be analyzed became few, return to RTL and use the serial simulation. This idea is shown in Figure 8.4.

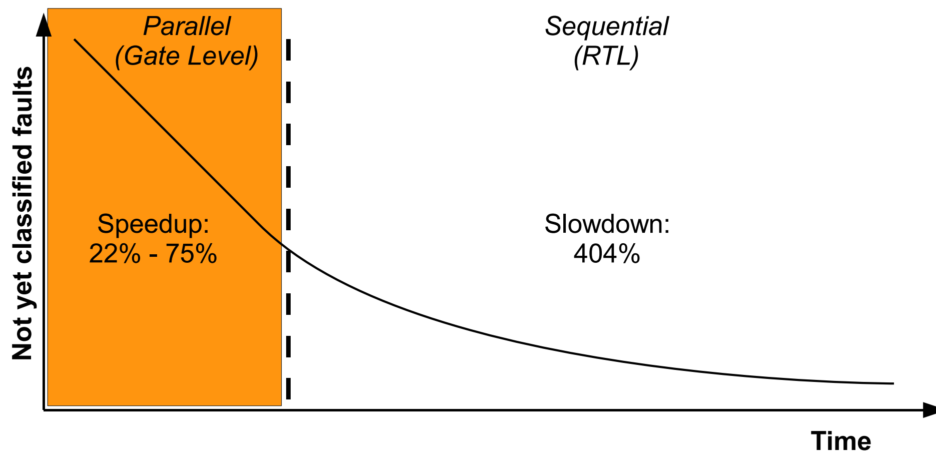


Fig. 8.4. Fault detection.

8.5 Concluding remarks

This work analyzes the main issues arising to implement an efficient parallel simulation engine for functional faults. Moreover, the set of implemented optimizations used to further increase performances of parallel simulation has been described, Experimental results show that even if the parallelization requires gate-level simulation, which is generally slower than RTL simulation, the overall simulation time is reduced as long as there are enough faults to be checked in parallel. Considering the results of the experiments, it seems that parallelization truly provides benefits to verification. Future work could explore further optimizations and, maybe, this will lead to parallelization being successfully adopted for industrial designs.

The work described in this chapter appears in the following publication: [49].

9

HIF Suite

From the modeling point of view, nowadays, it is common practice to define new systems by reusing previously developed components, that can be possibly modeled at different abstraction levels such as transaction-level modeling (TLM) and register-transfer level (RTL) by means of different Hardware Description Languages (HDL's) like VHDL, SystemC, Verilog, etc.. Such heterogeneity requires to either use co-simulation and co-verification techniques [21], or to convert pieces of code from different HDL's into an homogeneous description [44]. However, co-simulation techniques slow down the overall simulation, and prevents the use of tools not supporting a specific HDL. On the other hand, manual conversion from an HDL representation to another, as well as manual abstraction/refinement from an abstraction level to another, are not valuable solutions, since they are error-prone and time consuming tasks. Thus, both co-simulation and manual refinement reduce the advantages provided by the adoption of a reuse-based design methodology.

To avoid such problems, this chapter presents HIFSuite, an integrated set of tools and APIs for reusing already developed components and verifying their integration into new designs. Relying on the HIF language, HIFSuite allows system designers to convert HW/SW design descriptions between different HDL's and to manipulate them in an uniform and efficient way. In addition, from the verification point of view, HIFSuite is intended to provide a single framework that efficiently supports many fundamental activities like transactor-based verification [28, 101], mutation analysis [24], automatic test pattern generation, etc. Such activities generally require that designers and verification engineers define new components (e.g., transactors), or modify the design to introduce saboteurs [100], or represent the design by using mathematical models like extended finite state machines (EFSM) [108]. To the best of our knowledge, there are no tools in the literature that integrate all the previous features in a single framework. HIFSuite is intended to fill in the gap.

The author of this thesis have contributed to the development of HIFSuite addressing these issues:

1. Definition of HIF Semantics.
2. Definition of HIF syntactic constructs.
3. Definition and improvement of the HIF manipulation API's.
4. Mapping between HDL's and HIF syntactic constructs.
5. Implementation of HIF core library.

6. Implementation of some translation tools (sf2hif, hif2vhdl, hif2sc).
7. Optimization of the HIF core library.
8. Support to other developers.

From the site `http://hifsuite.edalab.it/` it is possible to download an HIFSuite demo. This chapter explains the main features of the HIFSuite, of the HIF language, and of translation tools, whilst manipulation tools are just summarized, since the author of this thesis has not collaborated to their implementation, and they are outside the scope of this thesis.

The chapter is organized as follows. Related work is described in Section 9.1. An overview of the main features of HIFSuite is presented in Section 9.2, while the HIF core-language is described in Section 9.3. The HIF-based conversion tools are presented in Section 9.4. Finally, remarks are discussed in Section 9.5.

9.1 Related Work

The issue of automatic translation and manipulation of HDL code has been addressed by different works.

VH2SC [93] is a translator utility from VHDL 87/93 to SystemC. It is not able to handle large designs, and presents some known bugs. It is no more maintained and the author itself suggest that it is not suited for industrial designs.

Another tool for automatically convert VHLD into SystemC is VHDL-to-SystemC-Converter [161]. However, it is limited to only to RTL synthesizable constructs.

Some approaches provide the capability of converting HDL code into C++ to increase simulation performances. A methodology to translate synthesizable Verilog into C++ has been implemented in the VTOC [153] tool. The methodology is based on synthesis-like transformations, which is able to statically resolve almost all scheduling problems. Thus, the design simulation switches from an event-driven to a cycle-based-like algorithm.

VHDLC [133] is a VHDL to C++ translator, which aims at fully VHDL'93 compliance. The project is at alpha stage, and thus, it is currently not suited for industrial applications.

The DVM [10] tool translates VHDL testbenches into C++. Testbenches are encompassed into a small simulable kernel which runs interconnected with the board. Running native C++ code avoids the overhead introduced by using an HDL simulator. Such a tool is restricted to only VHDL testbenches.

Verilator [164] is a Verilog simulator. It supports Verilog synthesizable and some PSL SystemVerilog and Synthesis assertions. To optimize the simulation, Verilator translates the design into an optimized C++ code, wrapped by a SystemC module. To achieve better performances, Verilator performs semantics manipulations which are not standard Verilog compliant. Thus, Verilator is meant to be used just as a simulator.

FreeHDL [53] is a simulator for VHDL, designed to run under Linux. The aim of the project is to create a simulator usable also with industrial designs. To improve performances, FreeHDL uses an internal tool, namely FreeHDL-v2cc, which translates the original VHDL design into C++ code. Thus, FreeHDL presents limitations similar to Verilator.

All previous approaches are not based on an intermediate format, and they target only a point to point translation from one HDL to another HDL or C++. Thus, manipulation

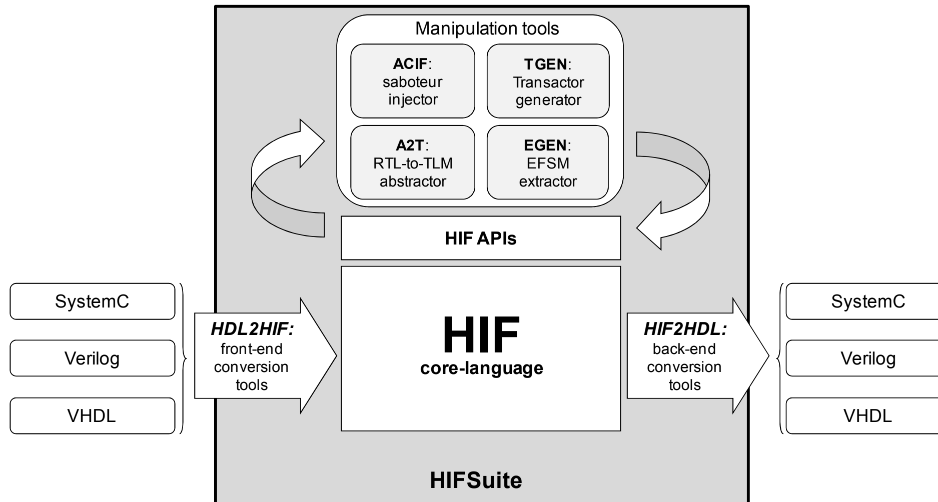


Fig. 9.1. HIFSuite overview.

on the code before the translation is not supported. On the contrary, some works have been proposed which use an intermediate format to allow manipulation of the target code for simplifying design manipulation and verification. In this context, AIRE/CE [110], previously known as IIR, is an Object-Oriented intermediate format. A front-end parser, namely SAVANT, is available to translate VHDL designs into such a language. By using AIRE/CE API's it is possible to develop verification and manipulation tools for VHDL designs. In a previous work [7], SAVANT was extended to allow the translation of VHDL designs into SystemC. Unfortunately, the AIRE/CE language is strictly tailored for VHDL code, and thus, it has been not easy to extend the SAVANT environment for supporting other HDLs, and particularly SystemC TLM.

To the best of our knowledge there is not a single comprehensive environment which integrates conversion capabilities from different HDLs at different abstraction levels and a powerful API to allow the development of manipulation tool required during the refinement and verification steps of a design.

The proposed HIF language has been designed to overcome limits and restrictions of previous works. In particular, HIF and the corresponding HIFSuite has been developed to address the following aspects:

1. supporting translation of several HDLs at both RTL and TLM level.
2. providing an object oriented set of APIs for fast and easy implementation of manipulation and verification tools.

9.2 HIFSuite Overview

Figure 9.1 shows an overview of the HIFSuite features and components. The majority of HIFSuite components have been developed in the context of three European Projects (SYMBAD, VERTIGO and COCONUT). HIFSuite is composed of:

- An *HIF core-language:* a set of HIF objects corresponding to traditional HDL constructs like, for example, processes, variable/signal declarations, sequential and concurrent statements, etc. (see Section 9.3).
- A set of front/back-end conversion tools (see Section 9.4):
  - *HDL2HIF*. Front-end tools that parse VHDL, Verilog and SystemC (RTL and TLM) descriptions and generate the corresponding HIF representations.
  - *HIF2HDL*. Back-end tools that convert HIF models into VHDL, Verilog or SystemC (RTL and TLM) code.
- A set of APIs that allow designers to develop HIF-based tools to explore, manipulate and extract information from HIF descriptions (see Section 9.3.3). The HIF code manipulated by such APIs can be converted back to the target HDLs by means of *HIF2HDL*.
- A set of tools developed upon the HIF APIs that manipulate HIF code to support modeling and verification of HW/SW systems, such as:
  - *EGEN*: a tool developed upon the HIF APIs that extracts an abstract model from HIF code. Such a tool, namely *EGEN*, automatically extracts EFSM models from HIF descriptions. EFSMs represent a compact way for modeling complex systems like, for example, communication protocols [105], buses [172] and controllers driving data-paths [87, 154]. Moreover, they represent an effective alternative to the traditional finite state machines (FSMs) for limiting the state explosion problem during test pattern generation [88].
  - *ACIF*: a tool that automatically injects saboteurs into HIF descriptions. Saboteur injection is important to evaluate dependability of computer systems [92]. In particular, it is a key ingredient for verification tools that relies on fault models like, for example, automatic test pattern generators (ATPGs) [18], tools that measure the property coverage [59] or evaluate the quality of testbenches through mutation analysis [36], etc.
  - *TGEN*: a tool that automatically generates transactors. Verification methodologies based on transactors allow an advantageous reuse of testbenches, properties and IP-cores in TLM-RTL mixed designs, thus guaranteeing a considerable saving of time [16]. Moreover, transactors are widely adopted for the refinement (and the sub-sequence verification) of TLM descriptions towards RTL components [40].
  - *A2T*: a tool that automatically abstracts RTL IPs into TLM models. Even if transactors allow designers to efficiently reuse RTL IPs at transaction level, mixed TLM-RTL designs cannot always completely benefit of the effectiveness provided by TLM. In particular, the main drawback of IP reuse via transactor is that the RTL IP acts as a bottleneck of the mixed TLM-RTL design, thus slowing down the simulation of the whole system. Therefore, by using *A2T*, the RTL IPs can be automatically abstracted at the same transaction level of the other modules composing the TLM design, to preserve the simulation speed typical of TLM without incurring in tedious and error-prone manual abstraction [25, 26].

The main features of HIF core-language are summarized in next Sections.

9.3 HIF Core-Language and APIs

HIF is an HW/SW description language structured as a tree of objects, similarly to XML. Each object describes a specific functionality or component that is typically provided by HDL languages like VHDL, Verilog and SystemC. However, even if HIF is quite intuitive to be read and manually written, it is not intended to be used for manually describing HW/SW systems. Rather, it is intended to provide designers with a convenient way for automatically manipulating HW/SW descriptions.

The requirements for HIF are manifold as it has to represent:

- system-level and TLM descriptions with abstract communication between system components;
- behavioral (algorithmic) hardware descriptions;
- RTL hardware descriptions;
- hardware structure descriptions;
- software algorithms.

To meet these requirements, HIF includes several concepts that are inspired by different languages. Concerning RTL and behavioral hardware descriptions, HIF is very much inspired to VHDL. On the other hand, some constructs have been taken from C/C++ programming language for the representations of algorithms (e.g., pointers and templates). The combination of these different features makes HIF a powerful language for HW/SW system representations.

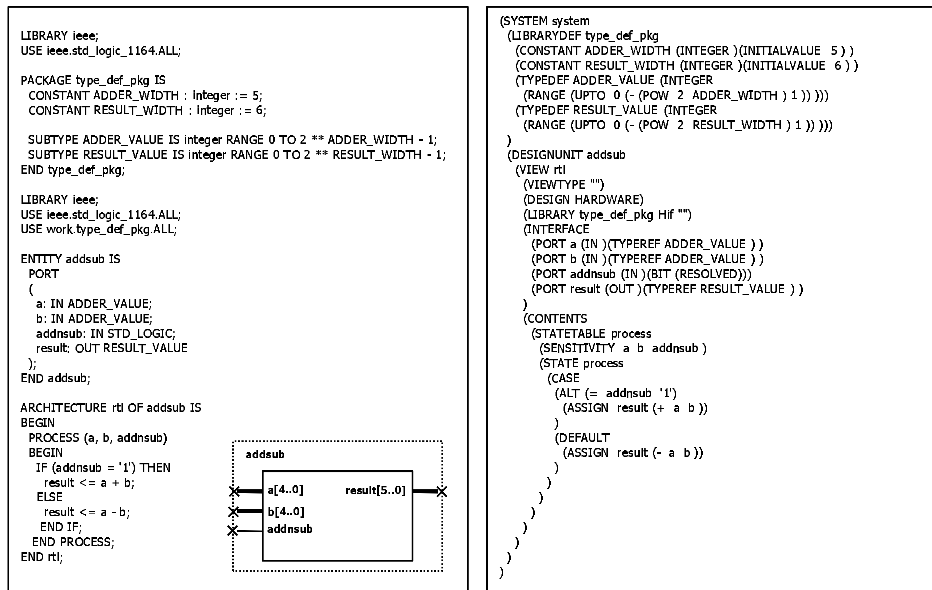
9.3.1 HIF Basic Elements

HIF is a description language structured as a tree of elements, similarly to XML (see Figure 9.2). It is very much like a classical programming language, i.e., a typed language which allows the definition of new types, and includes operations like assignments, loops, conditional executions, etc.. In addition, since HIF is intended to represent hardware descriptions, it also includes typical low-level HDL constructs (e.g., bit slices). Finally, concerning the possibility of structuring a design description, HIF allows the definition of components and subprograms.

To look at similarities between HIF and traditional HDLs, let us consider Figure 9.2. On the left side, a two-input parameterized adder/subtractor VHDL design is shown, while the corresponding HIF representation generated by the front-end tool *HDL2HIF* (see Section 9.4) is depicted on the right.

As a special feature, HIF gives the possibility to add supplementary information to language constructs in form of so-called *properties*. A list of properties can be associated to almost every syntactic constructs of the HIF language. Properties allow designers to express information for which no syntactic constructs are included in the HIF grammar, and therefore they give a great flexibility to the HIF language. For example, the fact that a signal `s` has to be considered as a clock signal can be expressed by adding a property `signal_type` to the signal declaration as follows:

```
(SIGNAL s (BIT) (PROPERTY signal_type clock)).
```



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE type_def_pkg IS
  CONSTANT ADDER_WIDTH : integer := 5;
  CONSTANT RESULT_WIDTH : integer := 6;

  SUBTYPE ADDER_VALUE IS integer RANGE 0 TO 2 ** ADDER_WIDTH - 1;
  SUBTYPE RESULT_VALUE IS integer RANGE 0 TO 2 ** RESULT_WIDTH - 1;
END type_def_pkg;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.type_def_pkg.ALL;

ENTITY addsub IS
  PORT
  (
    a: IN ADDER_VALUE;
    b: IN ADDER_VALUE;
    addnsub: IN STD_LOGIC;
    result: OUT RESULT_VALUE
  );
END addsub;

ARCHITECTURE rtl OF addsub IS
BEGIN
  PROCESS (a, b, addnsub)
  BEGIN
    IF (addnsub = '1') THEN
      result <= a + b;
    ELSE
      result <= a - b;
     END IF;
   END PROCESS;
END rtl;
```



```
(SYSTEM system
  (LIBRARYDEF type_def_pkg
    (CONSTANT ADDER_WIDTH (INTEGER )(INITIALVALUE  5 ) )
    (CONSTANT RESULT_WIDTH (INTEGER )(INITIALVALUE  6 ) )
    (TYPEDEF ADDER_VALUE (INTEGER
      (RANGE (UPTO  0 (- (POW  2  ADDER_WIDTH ) 1 )) )))
    (TYPEDEF RESULT_VALUE (INTEGER
      (RANGE (UPTO  0 (- (POW  2  RESULT_WIDTH ) 1 )) )))
  )
  (DESIGNUNIT addsub
    (VIEW rtl
      (VIEWTYPE "")
      (DESIGN HARDWARE)
      (LIBRARY type_def_pkg Hif "")
      (INTERFACE
        (PORT a (IN )(TYPEREF ADDER_VALUE ) )
        (PORT b (IN )(TYPEREF ADDER_VALUE ) )
        (PORT addnsub (IN )(BIT (RESOLVED)))
        (PORT result (OUT )(TYPEREF RESULT_VALUE ) )
      )
      (CONTENTS
        (STATETABLE process
          (SENSITIVITY  a  b  addnsub )
          (STATE process
            (CASE
              (ALT (=  addnsub  '1')
                (ASSIGN  result (+  a  b ))
              )
              (DEFAULT
                (ASSIGN  result (-  a  b ))
              )
            )
          )
        )
      )
    )
  )
)
```

Fig. 9.2. (a) A VHDL design description. (b) The textual format of the corresponding HIF representation.

9.3.2 System Description by using HIF

The top-level element of a system represented by an HIF description is the `SYSTEM` construct (see Figure 9.2(b)). It may contain the definition of one or more *libraries* which define new data types, constants and subprograms, and the description of *design units*. An HIF description may also contain a list of *protocols*, which describe communication mechanisms between design units.

Design units are modeled by `DESIGNUNIT` objects, which define the actual components of the system. A design unit may use types, constants and subprograms defined in libraries included in the `SYSTEM` construct.

The same design unit can be modeled in different ways inside the same system by using *views*. For example, different views of the same design unit can be modeled at different abstraction levels. Thus, a `VIEW` object is a concrete description of a system component. It includes the definition of an `INTERFACE` by which the component communicates with the other parts of the system. Moreover, a view may include libraries and local declarations. The internal structure of a view is described in details by means of the `CONTENTS` construct. To make a comparison with VHDL, a *view* can be seen as a generalization of VHDL *entity* and *architecture*.

An `INTERFACE` object gives the link between a design unit and the rest of the system. An *interface* can contain ports, and parameters.

A `CONTENTS` object can contain a list of local declarations, a list of state tables, which describe sequential processes, and a list of component instances and nets which connect such instances. Furthermore, a `CONTENTS` object can contain a set of *concurrent*

actions (called GLOBALACTIONs), i.e., assignments and procedure calls which assign values to a set of signals in a continuous manner.

Sequential Processes

in HIF, behaviors described by sequences of statements (i.e., processes) are expressed by *state tables*. A STATETABLE object defines a process, whose main control structure is an EFSM, and the related sensitivity list. The state tables can describe synchronous as well as combinational processes. The entry state of the state machine can be explicitly specified. Otherwise, the first state in the state list is considered as an entry state.

STATE objects included in the state table are identified by a unique name and they are associated to a list of instructions called *actions* (i.e., assignments, conditional statements, etc.) to be sequentially executed when the HIF model is converted into an HDL description for simulation.

Components Instances

descriptions where one or more components are instantiated and connected each other are modeled by using the INSTANCE and the NET constructs. An INSTANCE object describes an instance of a design unit. More precisely, an INSTANCE object refers to a specific view of the instantiated design unit.

A NET object contains a list of port references. Nets are used to express connectivity between interface elements of different design unit instances (i.e., system components).

Concurrent actions

they correspond to concurrent assignments and concurrent procedure calls of VHDL, and they are modeled by GLOBALACTION objects. Concurrent assignments are used to assign a new value to the target (which must be a signal or a port) each time the value of the assignment source changes. Similarly, concurrent procedure calls are used to assign a new value to signals mapped to the output parameters each time one of the input parameters changes its value.

Support for TLM constructs

TLM is becoming an usual practice for simplifying system-level design and architecture exploration. It allows the designers to focus on the design functionality while abstracting away implementation details that will be added at lower abstraction levels. The HIF language supports the SystemC TLM constructs provided by OSCI [159], which mainly rely on C++ constructs such as *pointers* and *templates*. Figure 9.3 shows a typical TLM interface with socket channels for blocking and non-blocking calls in SystemC and the corresponding HIF representation. The SystemC interface definition exploits nested C++ templates, which are preserved in the HIF description. The HIF language provides two keywords to support templates: TYPETP and TYPETPASSIGN. TYPETP is used for declaration of objects of template type. Instead TYPETPASSIGN is used for instantiation of template object as shown in Figure 9.3(b). In HIF, the declaration of pointers is represented by using the POINTER object as follows: `(POINTER type {property})`.

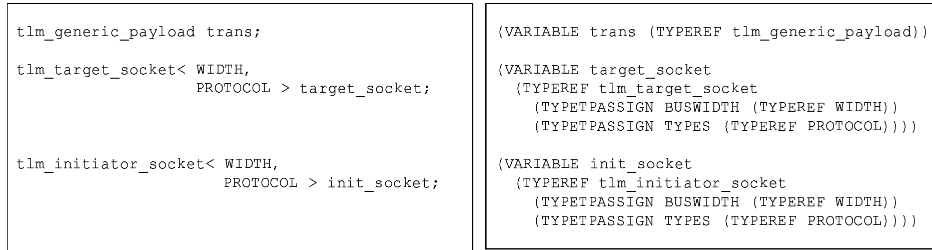


Fig. 9.3. (a) Example of TLM interface with socket channels for blocking and non-blocking calls in SystemC. (b) The corresponding HIF representation.

9.3.3 HIF Application Programming Interfaces

HIFSuite provides the HIF language with a set of powerful C++ APIs which allow to explore, manipulate and extract information from HIF descriptions. There are two different subsets in HIF APIs: the *HIF core-language APIs* and the *HIF manipulation APIs*.

HIF core-language APIs

each HIF construct is mapped to a C++ class that describes specific properties and attributes of the corresponding HDL construct. Each class is provided with a set of methods for getting or setting such properties and attributes.

For example, each assignment in Figure 9.2(b) is mapped to an `AssignObject` which is derived from `ActionObject` (see Figure 9.4). This class describes the assignment of an expression to a variable, a register, a signal, a parameter or a port, and it has two member fields corresponding to the left-hand side (target) and the right-hand side (source) of the assignment.

The UML class diagram in Figure 9.4 presents a share of the HIF core-language APIs class diagram. `Object` is the root of the HIF class hierarchy. Every class in the HIF core-language APIs has `Object` as its ultimate parent.

HIF Manipulation APIs

the HIF manipulation APIs are used to manipulate the objects in HIF trees.

The first step of any HIF manipulation consists of reading the HIF description by the following function:

```
Object* Hif::File::ASCII::read(const char* filename).
```

This function loads the file and build the corresponding tree data structure in memory. An analogous writing function allows to dump the modified HIF tree on a file:

```
char Hif::File::ASCII::write( const char* filename,
                              Object* obj);
```

Once the HIF file is loaded in memory, many APIs are available to navigate the HIF description. The most important are the following:

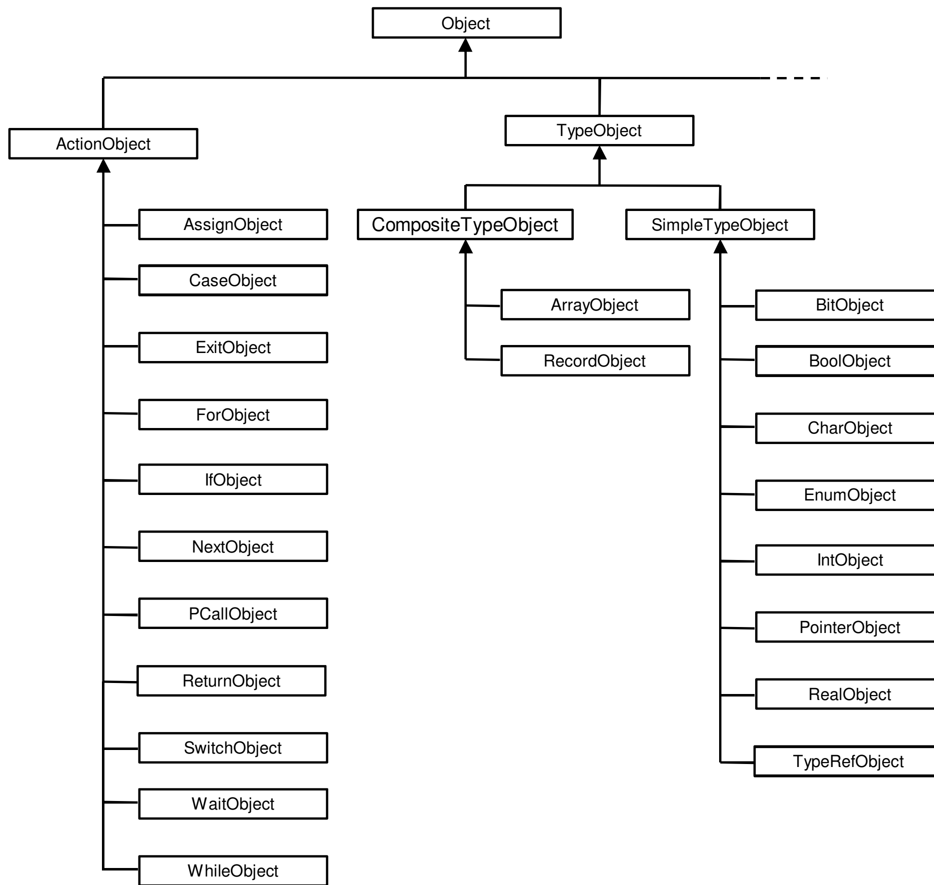


Fig. 9.4. A share of the HIF core language class diagram

```
Hif::hif_query query;
query.set_object_type(NameNode); //search for NameNode
query.set_name(state);           // search for string state
std::list<Node*> * found_object = Hif::search(base_object,query);
```

Fig. 9.5. Example of search function usage.

- *Search function.* The search function finds the objects that match a criteria specified by the user. It searches the target objects starting from a given object until it reaches the bottom (or the max depth) of the HIF tree. For example, the search function can be used to find out all variables that match the name `state` starting from `base_object`, as in Figure 9.5.
- *Visitor design pattern.* In object-oriented programming and software engineering, the visitor design pattern is generally adopted as a way for separating an algorithm from an object structure. A practical result of this separation is the ability to add new op-

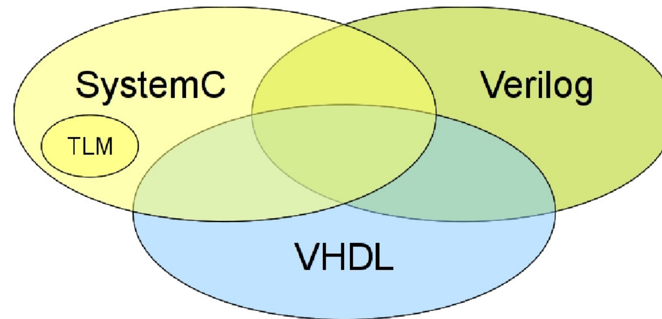


Fig. 9.6. Semantics relationships between different HDL's.

erations to existing object structures without modifying these structures. The visitor design pattern is very useful when there is a tree-based hierarchy of objects and it is necessary to allow an easy implementation of new features to manipulate such a tree. The HIF APIs provide visitor techniques in two forms: as an interface which must be extended to provide visitor operators, and as an `apply()` function. In the first case, a virtual method is inserted inside the HIF object hierarchy, which simply calls a specific-implemented visiting method on the object passed as parameter. The passed object is called *visitor* and it is a pure interface. The programmer has to implement such a visitor to visit and manage the HIF tree, by defining the desired visiting methods. In contrast, the `apply()` function is useful to perform an user-defined function on all the objects contained in a subtree of a HIF description. The signature for the apply function is the following:

```
void Hif::apply(Object *o,
                char(*f)(Object *,void *),
                void *data);
```

- *Compare function.* It provides designers with a way to compare two HIF objects and the corresponding subtrees. Its signature is the following:

```
static char compare (Object *obj1, Object *obj2)
```

- *Object replacement function.* It provides designers with a way for replacing an object and its subtree with a different object. Its signature is the following:

```
int Hif::replace(Object* from, Object* to)
```

9.3.4 HIF Semantics

Handling different HDLs that have different semantics by using a single intermediate language rises the importance of defining carefully a semantics for such intermediate language. In particular, the definition of a sound semantics is necessary for guaranteeing the correctness of the conversion and manipulation tools.

The semantics defined for HIF aims at supporting the representation of RTL designs for the main important and used HDLs (i.e., VHDL, Verilog, and SystemC) and the representation of TLM designs.

The main differences among VHDL, Verilog and SystemC semantics that make hard the automatic conversion of designs between them can be summarized as in the follows:

- *Data types*. Not all the languages have the same type management. Thus, the conversion between different languages requires to make explicit (or to remove) some cast or calls to type conversion functions.
- *Concurrent assignments*. The concurrent assignments of VHDL and Verilog do not have a direct mapping into a SystemC construct. They can be modeled in SystemC by converting each concurrent assignment into a concurrent process sensitive to the read signals and ports.
- *Operators*. The HDLs have different operators and different types on which such operators are defined. For example, VHDL uses the same operator symbol for both logic and bitwise operators, while Verilog and SystemC have different symbols for them.
- *TLM constructs*. SystemC allows TLM descriptions by using templates and pointers, while VHDL and Verilog support only RTL descriptions.
- *Variable declaration and scoping*. The behavior of variables and their scoping rules are different among HDLs. For example, in VHDL variables declared inside a process will retain the last assigned value between two subsequent process executions. In SystemC, a variable declared inside a process such as `SC_METHOD` will get the initial value at each new process invocation. To map the VHDL variable semantics into the SystemC context, the variable declaration should be moved outside the process and inside the module interface.
- *Statements*. Some programming techniques and statements cannot be directly mapped into another HDL. As an example, the SystemC pre and post increment operators are not valid in VHDL.

In this context, different solutions have been evaluated for adoption as HIF semantics (Figure 9.6):

- *RTL HDL specific semantics*. The semantics of an already existing RTL HDL (i.e., the VHDL or Verilog semantics) would be already well-defined and well-known. Nevertheless, this choice would be a restrictive solution since such languages do not apply to TLM descriptions.
- *SystemC semantics*. It would apply for both RTL and TLM designs. Nevertheless, SystemC is a *C++* library and, hence, its semantics corresponds to the *C++* semantics. It is not the best solution for implementing the back-end tools, as they would reduce the set of HIF constructs into the smaller set of HDL RTL constructs.
- *Union semantics*. It is the semantics obtained from the union of VHDL, Verilog and SystemC semantics. This solution would allow both TLM and RTL designs, and it also would simplify the translation from an HDL to HIF. On the other hand, it would require a greater effort for translating HIF descriptions to HDL descriptions, as not all constructs have an immediate mapping in every languages.
- *Intersection semantics*. It is obtained from the intersection of the Verilog, VHDL and SystemC semantics. It would simplify the translation from HIF to an HDL, as only constructs shared by all the HDLs would belong to HIF. Nevertheless, this choice would be too restrictive since only few RTL designs and no TLM description would be supported.
- *Dynamic semantics*. In this case HIF would not have a predefined semantics. Instead, each HIF description would keep track of the source HDL, thus importing also the se-

mantics of such HDL. This choice simplifies the translation from an HDL to HIF, but it implies a great effort in developing the back-end tools. Moreover, the HIF descriptions would be too complex for the manipulation tools since each tool should have a different behavior according to the design specific semantics.

For all these reasons, the semantics chosen for HIF has been the VHDL semantics enriched to support TLM constructs. The main advantages of such semantics are the following:

1. The intermediate language is strongly typed (like VHDL).
2. There is a simple RTL-to-RTL compatibility between different HDLs. The fact that VHDL is strongly typed, makes easier to map its semantics into other HDL semantics.
3. It supports TLM.

9.4 Conversion Tools

In this Section the main characteristics of the conversion tools are reported, by starting from an overview of the tool structures and, then, by showing the translation semantics such tools rely on.

9.4.1 The front-end and back-end conversion tools

The conversion tools are organized into front-end (*HDL2HIF*) and back-end (*HIF2HDL*) tool sets.

HDL2HIF converts HDL implementations into HIF. *HDL2HIF* supports conversions from VHDL, Verilog and SystemC, which are implemented in the submodules *VHDL2HIF*, *VERILOG2HIF* and *SC2HIF*,respectively.

The *VHDL2HIF* and *VERILOG2HIF* tools have a common structure, which is composed of the following modules:

- A pre-parsing module, which performs basic configuration operations and parameter parsing, and which selects the output format (readable plain text or binary).
- A parser based on GNU *Bison* [80], which directly creates an HIF-objects tree. The conversion process is based on a recursive algorithm that exploits a pre-ordered visit strategy on the syntax tree nodes.
- A post-conversion visitor, which refines the generated HIF tree according to the input language.
- A final routine, which dumps the HIF tree on a file.

The *SC2HIF* tool has the structure composed of the following modules:

- A pre-parsing module, which performs basic configuration operations and parameter parsing, and which selects the output format (readable plain text or binary).
- A parser based on GNU *Bison*, which creates an abstract syntax tree (AST) of the input code. Such an AST is composed of XML objects with dedicated tags.
- A core module, which converts the AST into an HIF-object tree. The conversion process is based on a recursive algorithm that exploits a pre-ordered visit strategy on the tree nodes.

Table 9.1. Matching of HDL types

HIF	VHDL	SystemC	Verilog
BIT	bit	sc_bit	wire or reg
BOOLEAN	boolean	bool	wire or reg
CHAR	char	char	
INTEGER	integer	int	integer
(ARRAY (PACKED) (INTEGER) (OF (BIT)) (RANGE))	bit_vector (RANGE)	sc_bv <RANGE>	wire[RANGE] or reg[RANGE]
(ARRAY (PACKED) (INTEGER) (OF (BIT (RESOLVED))) (RANGE))	std_logic_vector (RANGE)	sc_lv <RANGE>	wire[RANGE] or reg[RANGE]
BIT (RESOLVED)	std_logic	sc_logic	wire or reg
REAL	real	double (float)	real

- A post-conversion visitor, which refines the generated HIF tree.
- A final routine, which dumps the HIF tree on a file.

An intermediate XML tree has been preferred for translating SystemC descriptions to HIF, since the SystemC language is much more complex w.r.t. other HDLs,. The translation requires different checks in order to perform a correct mapping. This intermediate operation has been performed by using KaSCPar [86], an open source tool which has been improved to support TLM.

HIF2HDL converts HIF code back to VHDL (*HIF2VHDL*), Verilog (*HIF2VERILOG*) or SystemC (*HIF2SC*). The structure of HIF2HDL includes the following modules:

- A pre-parsing module, which sets up the conversion environment, performs basic configuration operations, parses the parameters, and sets the output language.
- A set of refinement visitors, which perform operations to allow an easier translation of HIF trees, according to the output language. For instance, in VHDL it is possible to specify the bit value *1* by writing `'1'`. On the other hand, in SystemC, `'1'` is interpreted as a character and, thus, a cast to the *sc_logic* type is required. To solve this problem, a visitor has been implemented to wrap the constant object `'1'` with an *sc_logic* cast object into the HIF tree.
- A module that dumps a partial conversion of the HIF code into temporary files. Such a module has been implemented to solve problems of consistency between the order adopted to visit the HIF AST and the order needed to print out the code in the target language. To avoid many complex checks, the tools firstly dump the output code directly in temporary files and then they merge the content of temporary files together in the correct order.
- A post-visit module, which merges together the temporary files and creates the final output.

In the following subsections, it is reported the implementation of most meaningful and critical HDL statements, their matching among VHDL, Verilog, and SystemC, and their representation in HIF. The HIFSuite conversion tools rely on these matching for converting designs among different HDL languages.

9.4.2 HDL types

Table 9.1 depicts the matching of the most important HDL types. In this work, all the VHDL types implemented into the VHDL IEEE libraries are considered as native VHDL

Table 9.2. Matching of HDL explicit cast

VHDL	HIF	SystemC	Verilog
unsigned(I)	(CAST I (UNSIGNED_TYPE (t_RANGE)))	to_unsigned(I, size)	$unsigned(I)
signed(I)	(CAST I (SIGNED_TYPE (t_RANGE)))	to_signed(I, size)	$signed(I)
std_logic_vector(I)	(CAST I (ARRAY (PACKED) (t_RANGE) (OF (BIT (RESOLVED)))))	sc_lv<size>(I)	

Table 9.3. Matching of HDL conversion functions

VHDL	HIF	SystemC	Verilog
to_unsigned(I)	(CONV I (UNSIGNED_TYPE (t_RANGE)))	to_unsigned(I, size)	$unsigned(I)
to_signed(I)	(CONV I (SIGNED_TYPE (t_RANGE)))	to_signed(I, size)	$signed(I)

types (e.g., the VHDL `std_logic_vector` and `std_logic`, which are defined inside the library *std_logic_1164* are considered native types).

In Table 9.1, the mapping of the Verilog types is not fully specified. In fact, it is possible to know the correct mapping into a `reg` or into a `wire` only during the conversion phase, by checking if the corresponding identifier is used as a signal or as a memory element.

Another issue about Verilog is that both *resolved* and *unresolved* logic types are mapped into the same Verilog constructs. This is forced by the fact that Verilog has only *resolved* types.

For the `INTEGER` type, it is worth to note that, in HIF, it is possible to specify a `RANGE` of values, the number of bits on which is represented, whether it is *signed* or *unsigned*, and whether the range is *upto* or *downto* (e.g., in VHDL a `natural` has a different range from SystemC `unsigned`).

9.4.3 HDL cast and type conversion functions

Every HDL languages have three possible type conversion methods:

- *Implicit cast*. The language allows to convert a general type into another. The translation is thus automatically performed by the compiler. As an example, in SystemC it is possible to implicitly cast a `char` to an `int`.
- *Explicit cast*. The language supports the type translation, even if it requires the designer to use a special language construct (i.e., a *cast*).
- *Conversion function*. The language does not support the type conversion and, thus, a manual conversion is required. To simplify the designer's task, many pre-defined conversion functions are usually supplied by supporting libraries (e.g., the VHDL IEEE library).

Since the HIF semantics is an extension of the VHDL semantics, the HIF type conversion rules are inherited from VHDL. As a consequence, since the implicit cast are not allowed in VHDL, they are not allowed neither in HIF. On the other hand, the explicit cast is represented by the `CAST` object, while the conversion functions are mapped into `CONV` objects. The set of conversion functions include all the conversion functions implemented into the VHDL IEEE library.

Tables 9.2 and 9.3 report the matching of some cast and conversion functions, when they are implemented in VHDL, Verilog, and SystemC and how they are represented in HIF. These assumtions hold:

- `I` is the generic expression on which the cast or conversion is performed.
- `t_RANGE` is a range as intended into the HIF syntax.
- `size` is the size in bits needed to represent the values in `t_RANGE`.

In Verilog there are only two casting directives (`$signed()` and `$unsigned()`), since it is a loosely typed language. Thus, it requires only conversion tasks from signed to unsigned types and vice versa.

9.4.4 HDL operators

Table 9.4. Translation of HDL operators

<table>
<tr><th>HIF</th><th>VHDL</th><th>SystemC</th><th>Verilog</th></tr>
<tr><td colspan="4">Arithmetic Operators</td></tr>
<tr><td>+</td><td>+</td><td>+</td><td>+</td></tr>
<tr><td>-</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>*</td><td>*</td><td>*</td><td>*</td></tr>
<tr><td>/</td><td>/</td><td>/</td><td>/</td></tr>
<tr><td>MOD</td><td>mod</td><td>(%)</td><td>%</td></tr>
<tr><td>CONCAT</td><td>&</td><td>(a, b)</td><td>{a, b}</td></tr>
<tr><td colspan="4">Bitwise Operators</td></tr>
<tr><td>&&</td><td>and</td><td>&</td><td>&</td></tr>
<tr><td>||</td><td>or</td><td>|</td><td>|</td></tr>
<tr><td>^^</td><td>xor</td><td>^</td><td>^</td></tr>
<tr><td>!!</td><td>not</td><td>~</td><td>~</td></tr>
<tr><td colspan="4">Logic Operators</td></tr>
<tr><td>|</td><td>or</td><td>||</td><td>||</td></tr>
<tr><td>&</td><td>and</td><td>&&</td><td>&&</td></tr>
<tr><td>!</td><td>not</td><td>!</td><td>!</td></tr>
<tr><td colspan="4">Comparison Operators</td></tr>
<tr><td>=</td><td>=</td><td>==</td><td>==</td></tr>
<tr><td>/=</td><td>/=</td><td>!=</td><td>!=</td></tr>
<tr><td><=</td><td><=</td><td><=</td><td><=</td></tr>
<tr><td><</td><td><</td><td><</td><td><</td></tr>
<tr><td>>=</td><td>>=</td><td>>=</td><td>>=</td></tr>
<tr><td>></td><td>></td><td>></td><td>></td></tr>
<tr><td colspan="4">Arithmetic Shift Operators</td></tr>
<tr><td>SLA</td><td>sla</td><td><<</td><td><<<</td></tr>
<tr><td>SRA</td><td>sra</td><td>>></td><td>>>></td></tr>
<tr><td colspan="4">Logic Shift Operators</td></tr>
<tr><td>SLL</td><td>sll</td><td><<</td><td><<</td></tr>
<tr><td>SRL</td><td>srl</td><td>>></td><td>>></td></tr>
</table>

The HIF language has a VHDL-like set of native operators with the exception that, in HIF, the difference between logic and bitwise operators is preserved. In addition, HIF has some operators that have not translation into VHDL or Verilog, as they are related to TLM designs (e.g., the pointer dereferencing operator, which is available only in SystemC).

Table 9.4 reports the matching among several operators. The shift operator is a meaningful example of operator conversion. The shift operator of Verilog is *arithmetic* if the operand is signed, otherwise it is logic. In contrast, the right shift semantics of C++ is platform dependent, since the *logic* or *arithmetic* shift is not specified by the standard. Thus, the mapping from SystemC to HIF is a platform-dependent code, and the equivalence cannot be guaranteed when converting HIF designs to SystemC. For this reasons, in this case, warnings are raised to the users by the conversion tools.

9.4.5 HDL structural statements

Table 9.5. Matching of HDL structural statements

Note	HIF	VHDL	SystemC	Verilog
1	DESIGNUNIT	entity, architecture	SC_MODULE	module
2	STATETABLE	process	SC_METHOD	always
3	ASSIGN	<= or :=	=	<= or =
4	IFGENERATE	if (cond) generate	#if (cond) concurrent_statement #endif	generate if (cond) concurrent_statement
5	FORGENERATE	for (cond) generate concurrent_statement end generate;	for (cond) { concurrent_statement }	generate for (cond) concurrent_statement end endgenerate
6	VARIABLE $id\ type$	VARIABLE $id : type$;	$type\ id$;	$type\ id$;

Table 9.5 shows how the structural statements are matched among HDLs. Note 1 indicates that in HIF each design unit can have one or more `VIEW` objects, each one containing one `INTERFACE` and one `CONTENTS` object. In contrast, in VHDL, it is possible to attach one or more architectures to a single interface. To achieve such behavior in HIF, many `VIEW` objects are created, each one having the same interface.

Note 2 is related to the description of a process into different HDLs. For SystemC, there are three kind of process constructs (i.e., `SC_METHOD`, `SC_THREAD` and `SC_CTHREAD`), while HIF has only one type of process (like VHDL). Thus, during the conversion from and to SystemC, the conversion tools recognize the SystemC process type and perform the code analysis for the correct mapping.

Note 3 is related to the management of assignments. There are two syntax for VHDL assignments (i.e., one for signals and one for variables) and two assignment operators in Verilog (i.e., blocking and continuous). In SystemC, a single assignment applies for both signals and variables.

Note 4 and 5 are related to constructs `FORGENERATE` and `IFGENERATE`. They are typical of VHDL and Verilog languages, while they have not a corresponding native construct in SystemC. Their conversion is achieved by inserting a loop (or a conditional statement) into the SystemC module constructor.

Note 6 is related to the syntax mapping for a variable declaration. In this case, a simple syntax-based translation is not enough since the declarations have different semantics depending on the HDL. This translation issue is addressed in detail in Section 9.4.6.

9.4.6 HDL declaration semantics

Converting declarations from different languages is challenging, because each HDL has different default initialization values, different scoping rules, different visibility, and different lifetime rules. As an example, a simple `int` in SystemC has not a default value, while in VHDL an `INTEGER` takes the leftmost value of the type range.

HIF, like VHDL, does not allow default values. Instead, each declaration have an explicit initialization value. In this way, there are not initialization problems when converting from HIF to another HDL. The HIFSuite front-end tools that translate from an HDL to HIF are demanded to recognize any declaration and to explicit the initialization value, according to the source HDL.

Table 9.6. Matching of SystemC and VHDL (HIF) declarations

SystemC	VHDL (HIF)
SC_MODULE	
Static constants	$CONSTANT$ declared and initialized inside the $architecture$
Input / output ports	$PORT$ declared inside the $entity$
Variable	SHARED VARIABLE declared inside the $architecture$
Static variable	*SHARED VARIABLE* declared inside the $architecture$
SC_METHOD	
Constant	$CONSTANT$ declared and initialized inside the $process$
Static constant	$CONSTANT$ declared and initialized inside the $process$
Variable	$VARIABLE$ declared and initialized inside the $process$
Static variable	$VARIABLE$ declared inside the $process$
SC_THREAD	
Constant	$CONSTANT$ declared and initialized inside the $process$
Static constant	$CONSTANT$ declared and initialized inside the $process$
Variable	$VARIABLE$ declared inside the $process$
Static variable	$VARIABLE$ declared inside the $process$

Table 9.7. Matching of Verilog and VHDL (HIF) declarations

VERILOG	VHDL (HIF)
Module	
parameter	added a $generic$ declaration inside the $entity$
localparam	$constant$ declared inside the $architecture$
input/output	$port$ declared into the $entity$
wire	$signal$ declared into the $architecture$
reg	$signal$ or $variable$ (code specific analysis required) declared inside the $architecture$

Table 9.8. Matching of VHDL (HIF) and SystemC or Verilog declarations

VHDL (HIF)	SystemC	Verilog
Entity		
Port	sc_in/out declared into SC_MODULE	$input/output$ declared at the beginning of the $module$
Shared variable	variable declared inside the module and initialized into the constructor	reg declared at the beginning of the $module$
Constant	*const* declared inside the class	$parameter$ declared at the beginning of the $module$
Signal	*sc_signal* declared inside the SC_MODULE	$wire$ declared at the beginning of the $module$
Architecture		
Constant	Static constant declared and initialized inside the SC_MODULE	$parameter$ declared and assigned inside the $module$
Shared variable	class variable declared inside the SC_MODULE	reg declared at the beginning of the $module$
Signal	sc_signal declared inside the SC_MODULE	$wire$ declared at the beginning of the $module$
Process		
Variable	Variable declared inside the SC_MODULE	reg declared before the process which uses it
Constant	constant declared inside the $process$	constant declared inside the $module$

For the sake of clarity, the matching of declarations between HDLs related to the front-end tools are separated from those related to the back-end tools. Considering the front-end tools, Table 9.6 reports the matching of the semantics between SystemC and VHDL declarations. VHDL and HIF declarations have the same semantics. Table 9.7 shows the matching between Verilog and VHDL (HIF) declarations.

Considering the back-end tools, Table 9.8 reports the matching between VHDL (HIF) and SystemC or Verilog declarations.

For allowing a correct conversion, the conversion tools can change the declaration scope (e.g., to have a correct lifetime). In this case, the translation tools automatically rename such a declaration and each of its occurrences, in order to avoid identifiers conflicts.

9.5 Concluding Remarks

This chapter has presented an overview of *HIFSuite*, a set of conversion and manipulation tools that rely on the HIF language. HIFSuite provides designers and verification engineers with the following features:

- *Conversion from HDLs to HIF and viceversa.* Current front-end and back-end tools support a great number of RTL VHDL, Verilog and SystemC constructs, and the core part of TLM SystemC. The extension for supporting further constructs is under development, and will be available soon. The HIF descriptions generated by the front-end tools are structured like syntax trees, thus it is easy to write algorithms that manipulate the nodes of the tree.
- *Merging of mixed HDL descriptions.* Systems described partially in VHDL or Verilog or SystemC, can be converted into the HIF representation, and then merged to obtain a final model implemented into an unique HDL.
- *Extendibility* The HIF library engine is structured to be easily extended. A special HIF object, called ProperyObject, is provided to describe non-standard or new features of other HIF objects.
- *HIF code manipulation*. A set of HIF-based manipulation tools are already available and they have been introduced in the previous sections. Such tools can be used into modeling or verification workflows that adopt different HDL languages. New tools can be easily implemented by means of a powerful APIs library, as they are implemented in C++.

The work described in this chapter appears in the following publications: [22, 23].

10

SystemC Network Simulation Library

Networked Embedded Systems (NES's) are devices characterized by their communication capabilities, like PDAs, cellphones, routers, wireless sensors and actuators. Their widespread use has generated significant research for their efficient design and integration into heterogeneous networks [46, 51, 65, 114, 144, 162].

Thus, system and Network design can be seen as two different aspects of the same design problem, but they are generally addressed by different people belonging to different knowledge domains and using different tools. Figure 10.1 shows the relationships between some modeling languages and simulation tools. In the context of HW design, the most popular languages are VHDL and Verilog. Such languages aim at providing simulable models. The focus is the functional description of computational blocks and the structural interconnection among them. Particular attention is given to the description of concurrent processes with the specification of the synchronization among them through wait/notify mechanisms. For SW design, one of the most widespread languages for embedded systems is C++, since it allows to use many modern design techniques, like object oriented programming and polymorphism, while providing high performances. SystemC [97] is a HDL implemented as a C++ library. It is gaining acceptance for its flexibility in describing both HW and SW components and for the presence of add-on libraries for Transaction-Level Modeling (TLM) and verification. In the context of network simulation, current tools reproduce the functional behavior of protocols, manage time information about transmission and reception of events and simulate packet losses and bit errors. Network can be modeled at different levels of detail, from packet transmission down to signal propagation [14, 30, 116].

The use of a single tool for System and Network modeling would be an advantage for the design of networked embedded systems.

Network tools cannot be used for System design since they do not model concurrency within each network node and do not provide a direct path to hardware/software synthesis.

A possible solution could be a Network/System cosimulation, by integrating the respective simulation kernels [66]. This approach could incur into inacceptable simulation time, since two simulation kernels, i.e. the Network and System simulators, shall run synchronized.

System languages and tools might be the right candidate for implementing an efficient System/Network simulation. In fact, as shown in Figure 10.2, HDL's already model com-

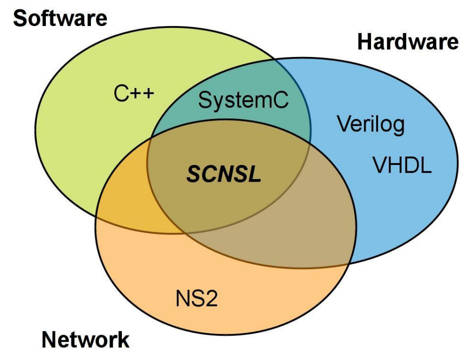


Fig. 10.1. SCNSL in the context of modeling languages and tools.

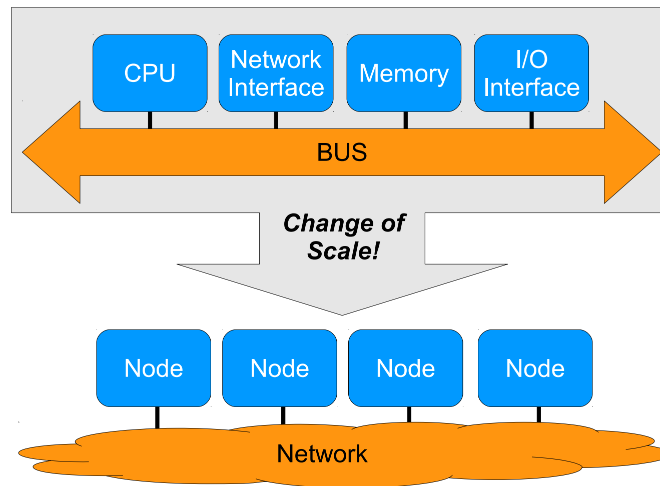


Fig. 10.2. Key idea: using HDL's to model communication between network nodes.

munication at system level, thus it seems feasable to extend them, to model also network communication.

Some works have addressed this issue, creating the communication channel as a bit-oriented model [150]. This kind of simulation can be very accurate, but it is also quite slow. This makes such tools more suitable during the last design phases, when reproducing low-level channel behaviors is crucial. Viceversa, traditional network simulators as NS-2 are designed at packet-level, to trade accuracy for simulation speed.

Thus, at early stages of design, it could be useful to have network simulators written by using HDL's, able to simulate at packet-level, to allow a fast verification of the NES's under design.

However, the use of a system description language for network modeling requires the creation of a basic set of primitives to simulate the asynchronous transmission of variable-length packets. To fill this gap, in this work the potentiality of SystemC has been exploited in the simulation of packet-switched networks, by implementing the *SystemC Network Simulation Library* (SCNSL). It has been developed as a SystemC extension library, in

order to be fully compatible with other SystemC optional libraries, like the AMS and Verification libraries. The simulator is freely available to the designers' community at `http://scnsl.sourceforge.net`.

In the past, SystemC was successfully used to describe network-on-chip architectures [45] and to simulate the lowest network layers of the Bluetooth communication standard [42].

In the proposed simulator, devices are modeled in SystemC and their instances are connected to a module that reproduces the behavior of the communication channel; propagation delay, interference, collisions and path loss are taken into account by considering the spatial position of nodes and their on-going transmissions. The design of tasks can be dealt at different abstraction levels: from electronic system level (e.g., Transaction Level Modeling) down to Register Transfer Level (RTL). After each refinement step, nodes can be tested in their network environment to verify that communication constraints are met. Tasks with different functionality or described at different abstraction levels can be mixed in the simulation thus allowing the exploration of network scenarios made of heterogeneous devices. Synthesis can be directly performed on those models provided that they are described by using a suitable subset of the SystemC syntax.

This chapter is organized as follows. Section 10.1 reviews past literature about System/Network co-simulation and co-design. Section 10.2 describes an overview of the SystemC Network Simulation Library. Section 10.3 outlines the main issues and solutions brought by this work, which motivate the main architectural choices made in SCNSL. Section 10.4 reports experimental results and, finally, conclusions are drawn in Section 10.5.

10.1 Related work

In the last years a lot of network simulators have been implemented, addressing different issues.

NS-2 [116] is a packet-oriented simulator, developed originally for wired networks and then extended to the wireless case. Even though it is widely adopted, it suffers of some problems. First of all its internal software architecture is not well structured, making extension and maintenance quite difficult. Moreover, code implementing node functionality can hardly be re-used on actual systems since it is strongly affected by the simulator internal organization.

TOSSIM [134] is a simulation tool specifically tailored for wireless sensor nodes running TinyOS operating system. This tool emulates the same SW code which is executed by actual devices and, thus, it provides accurate results. Furthermore, an extension, named PowerTOSSIM [163], allows also to evaluate power consumption. However its scope of application is limited to platforms supporting TinyOS.

Several other solutions have been proposed for the joint modeling of System and Network. An actual network is included in the simulation loop in [144]. The network and hardware co-simulation is addressed by allowing the HW description language to directly access the host communication primitives. As pointed out by the authors, this approach avoids some design errors that can occur if using a simplified and abstract network model, e.g., timeouts of lower level protocols. The use of the actual network restricts the set of available protocols to those provided by the host and, more important, the creation of complex topologies might be difficult and expensive. Finally, the synchronization between simulation time and actual time is not addressed.

SystemC has been used to model the lower levels of the Bluetooth protocol in [42]. The objective was to analyze performance at behavioral level and to assess power consumption. Despite the interesting results achieved, the work does not provide a general strategy for SystemC-based network simulation.

A wireless channel model has been implemented in [8]. The proposed methodology exploits TLM to gain a fast simulation, and it is able to reproduce some low level communication errors. The noise mechanism is based on statistical basis, and the channel modeling follows a peer-to-peer like structure. This leads to the creation of un-shared channels without collision mechanisms, which seems a rough approximation of actual behavior of a wireless channel which is by definition a shared channel with collisions.

SystemClick [147] is a framework at Electronic System Level, based on SystemC TLM, tailored for design space exploration of WLAN stations and networks. It integrates a performance profiler, and automatically generates SystemC network models starting from Click [107] descriptions. The framework allows only TLM-based exploration, and do not provides standard pre-built protocols, reducing its applicability. Moreover, wired networks are not addressed.

A bit-accurate C simulator of the WCDMA core of 3GPP terminals has been proposed in [114]. Its experimental results show that with this approach it is possible to achieve a good design speedup, but the work does not suggest a general and reusable approach to perform network simulations.

SystemC and MATLAB integration has been explored in [162]. MATLAB is used to specify the system at the highest level of abstraction, and it is used as golden model for the refinement of a SystemC implementation. The use of the MATLAB scripting language simplifies the specification of the requirements with respect to a more complex language such as C++/SystemC. In the paper, only the design of hardware and software components are taken into account, not considering the network as a true design space dimension. For example, the simulation of the system under design in a network context seems missing.

A C++ methodology and an environment to model and co-simulate hardware and software components has been proposed in [46]. The paper is focused on the design of the digital part of an ADSL modem. The proposed environment allows a simulation which reproduces the timing behavior and obtains a good simulation speed. The environment is suitable to implement a device and some related software, but it does not seems enough versatile for verifying the device under development with complex network models and with different communication protocols.

The integration of an instruction set simulator with a simulator of the formal Specification Description Language (SDL) is presented in [51]. As case study, the implementation of a wireless MAC layer is described. This methodology is interesting for requirements verification, but it lacks the ability to refine the design into a complex network scenario.

Summarizing, the past approaches have the following gaps:

- Some works are focused only on the design space exploration of the network, lacking the integration with System design.
- Some works address System design, without considering the network as a dimension of the design space.
- Some works integrate both System and Network design, but their applicability is restricted to specific fields, like wireless scenarios, or specific protocols.

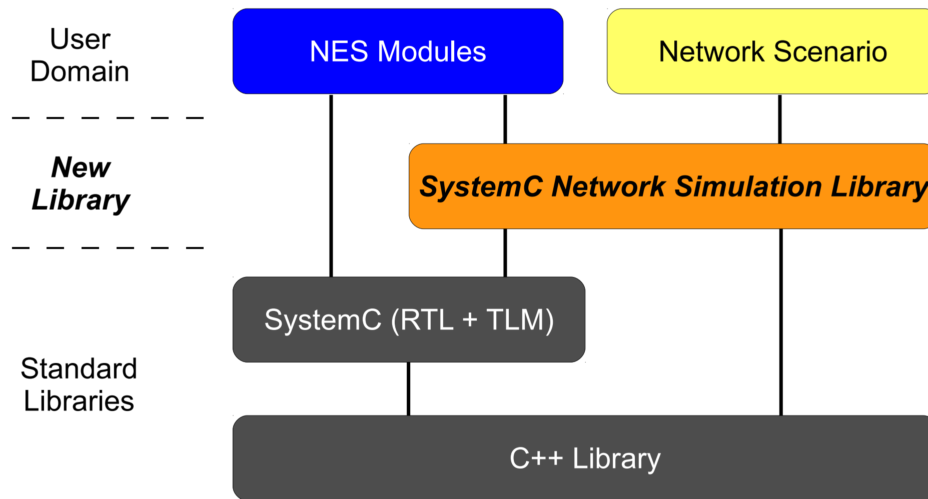


Fig. 10.3. SCNSL in the context of SystemC modeling.

In this work, a new network simulator is described, namely the SystemC Network Simulation Library (SCNSL), which tries to overcome these limits, thanks to the high flexibility of SystemC.

10.2 SCNSL architecture

The driving motivation at the base of SCNSL is to have a single simulation tool to model both the embedded systems under design and the surrounding network environment. SystemC has been chosen for its great flexibility, but a lot of work has been done to introduce some important elements for network simulation, as described in Section 10.3.

Figure 10.3 shows the relationship among the system under design, SCNSL, the SystemC standard library and the C++ library. In traditional scenarios, the system under design is modeled by using the primitives provided by the SystemC standard library, i.e., modules, processes, ports, and events. The resulting module is then simulated by a simulation engine, either the one provided in the SystemC free distribution or a third-party tool. To perform network simulations new primitives are required. Starting from SystemC primitives, SCNSL provides such elements so that they can be used together with System models to create network scenarios. Thus, SCNSL can be considered as an optional library for SystemC designs, like, for example, the SystemC Verification Library, and it is fully compatible with other optional libraries.

Another point regards the description of the simulation scenario. In SystemC, such description is usually provided in the `sc_main()` function which creates module instances and connects them before starting simulation; in this phase, it is not possible to specify simulation events as in a story board (e.g., "at time X the module Y is activated"). Instead, in many network simulators such functionality is available and the designer not only specifies the network topology, but also can plan events, e.g., node movements, link failures, activation/de-activation of traffic sources, and packet drops. For this reason, SCNSL also supports the registration of such network-oriented events during the initial instantiation

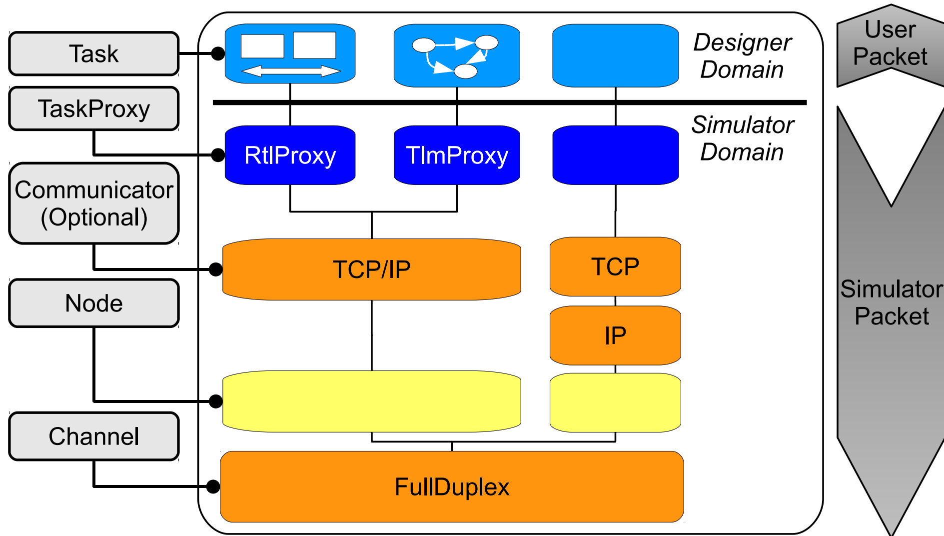


Fig. 10.4. Main components of SCNSL.

of SystemC modules. Moreover, since the description of a network scenario can involve thousands of components, SCNSL provides high level factories to simplify their instantiation.

As depicted in Figure 10.3, the model of the system under design uses both traditional SystemC primitives for the specification of its internal behavior, and SCNSL primitives to send and receive packets on the network channel and to test if the channel is busy. SCNSL takes in charge the translation of network primitives (e.g., packets events) into SystemC primitives.

10.2.1 Main components

Figure 10.4 shows the main components of SCNSL.

Tasks are the application functionalities which are under development. Thus, tasks shall be implemented by designers either at RTl or TLM level. From the point of view of a network simulator, a task is just the producer or consumer of packets and therefore its implementation is not important. However, for the system designer, task implementation is crucial and many operations are connected to its modeling, i.e., change of abstraction level, validation, fault injection, HW/SW partitioning, mapping to an available platform, synthesis, and so forth. For this reason the class `TaskProxy_if_t` has been introduced, which decouples task implementation from the backend which simulates the network. Each Task instance is connected to one or more TaskProxy instances and, from the perspective of the network simulation kernel, the TaskProxy instance is the alter-ego of the task. Viceversa, from the point of view of the application, each TaskProxy can represent a sort of socket interface, since it provides the primitives for network communication. This solution allows to keep a stable interface between the TaskProxy and the simulation kernel and, at the same time, to let complete freedom in the modeling choices for the task, e.g., interconnections of basic blocks or finite-state machines. Two different

TaskProxy interfaces are provided depending on the abstraction level of the connected Task (i.e., RTL or TLM). It is worth noting that other SystemC libraries can also be used in task implementation, e.g., re-used IP blocks and testing components such as the well-known SystemC Verification Library. These libraries may simplify designer's work even if they are outside the scope of network simulation.

Tasks are hosted on *Nodes*, which are the abstraction of physical devices. Thus, tasks deployed on different nodes shall communicate by using the API provided by SCNSL for the network communication, while tasks deployed on the same node shall communicate by using standard SystemC communication primitives.

The `Channel_if_t` class represents the network simulation kernel interface. A channel is the abstraction of the transmission medium, and thus it shall simulate eventual communication errors, like packet collisions. Standard SystemC channels are generally used to model interconnections between HW components and, therefore, they can be used to model network at physical level [42]. However, many general purpose network simulators reproduce transmissions at packet level to speed up simulations. SCNSL follows this approach, providing models for wired (full-duplex, half-duplex and unidirectional) and wireless channels.

Communication protocols are a key concept for a network simulation. Users could require to implement protocols ex-novo, in case protocols are under design, or they could rely on protocol models provided by the simulator. To allow maximum flexibility, SCNSL supports both these possibilities. In particular, users can implement protocols inside tasks. On the other hand, standard protocol models are provided in optional backend components, namely *Communicators*. A communicator is a SCNSL component implementing the interface `Communicator_if_t`. New capabilities and behavior can be easily added by extending this class. Communicators can be interconnected each other to create chains. Each valid chain shall have on one end a TaskProxy instance and, on the other end, a Node; hence transmitted packets will move from the source TaskProxy to the Node, traversing zero or more intermediate communicators, then they shall be transmitted by using a channel model, and finally they will eventually traverse the communicators placed between the destination node and the destination TaskProxy. In this way, it is possible to modify the simulation behavior by just creating a new communicator and placing its instance between taskproxies and nodes. Communicators can be used to implement not only protocols, but also buffers to store packets, simulation tracing facilities, etc.

Another critical point in the design of the tool has been the concept of *packet*. Generally, packet format depends on the corresponding protocol even if some features are always present, e.g., the length and source/destination addresses. System design requires a bit-accurate description of packet contents to test parsing functionality while from the point of view of the network simulator the strictly required fields are the length for bitrate computation and some flags to mark collisions (if routing is performed by the simulator, source/destination addresses are used too). Furthermore, the smaller the number of different packet formats, the more efficient is the simulator implementation. To meet these opposite requirements in SCNSL, an internal packet format is used by the simulator while the system designer can use other different packet formats according to protocol design. The conversion between the user packet format and the internal packet format is performed in the TaskProxy.

10.3 Issues in implementing SCNSL

Some issues encountered during the development of SCNSL have been:

- **Tasks & Nodes** (Section 10.3.1): System main components are processes, grouped into modules. Network scenario main components are tasks, i.e. functionalities, grouped into nodes.
- **Transmission validity** (Section 10.3.2): a packet could arrive invalid to the receiver, with reference to collision and out-of-range transmissions.
- **RTL simulation** (Section10.3.3): RTL models are usually bit-oriented, while the simulation is performed at packet-level.
- **Packets** (Section 10.3.4): SCNSL requires a standard packet to perform correctly the transmission simulation, but a packet is different from a RTL signal or a TLM payload.
- **TLM & RTL** (Section 10.3.5): the co-existence of RTL and TLM models during the same simulation requires the creation of a sort of transactor.
- **Topology** (Section 10.3.6): a network scenario is characterized by the concept of topology, i.e. the location of nodes in the physical space. Moreover the scenario could change dynamically during the simulation. Viceversa, System modules are organized into hierarchies, which cannot change during the simulation.
- **Communication protocols and channels** (Section 10.3.7): System module communication is standardized, since RTL modules shall communicate by using ports and signals, and TLM modules will communicate by using a standard payload and standard communication interfaces and algorithms. Network communication is more complex, since each scenario can use different protocols and channels.

The remaining of this Section addresses these issues and describes the solution adopted in SCNSL.

10.3.1 Tasks and nodes

From the point of view of network simulation, the main entities are tasks, which are the producers and consumers of packets, and nodes, which creates the network topology.

Since tasks represents the functionality of the application, their implementation shall be performed by SCNSL users. Tasks can represent both HW and SW components, and thus, it is possible to use both RTL and TLM abstraction levels. Anyway, SCNSL implements some standard tasks, which are common in network simulators. For instance, a *Constant Bit Rate* task is useful to create some interfering network traffic during a simulation, to test the application in case of packet losts. In SCNSL, each task shall implement a standard interface, `RtlTask_if_t` or `TlmTask_if_t`, according with the task abstraction level.

Nodes are the abstraction of physical devices. Thus, they contain physical related properties, like the transmission bitrate of the network interface, spatial position, etc. Each node can contain one or more tasks, and can be bounded to one or more physical channels. Since SCNSL assumes that the components under design are implemented as tasks, in SCNSL a general implementation of a node is provided by the backend, in the `Node_t` class.

10.3.2 Transmission validity assessment

In wireless and half-duplex scenarios an important feature of the network simulator kernel is the assessment of transmission validity which could be compromised by collisions and out-of-range distances. The validity check has been implemented by using two flags and a counter. The first flag is associated to each node pair and it is used to check the validity of the transmission as far as the distance is concerned; if the sender or the receiver of an ongoing transmission has been moved outside the maximum transmission range, this flag is set to false. The second flag and the counter are associated to each node and they are used to check the validity with respect to collisions. The counter is used to register the numbers of active transmitters which are interfering at a given receiver; if the value of this counter is greater than one, then on-going transmission to the given receiver are not valid since they are compromised by collisions. However, even if, at a given time, the counter holds one, the transmission could be invalid due to previous collisions; the flag has the purpose to track this case. When a packet transmission is completed, if the value of the counter is greater than one, the flag is set to false. The combined use of the flag and the counter allows to cover all transmission cases in which packet validity is compromised by collisions.

10.3.3 Simulation of RTL models

As said before, SCNSL supports the modeling of tasks at different abstraction levels. In case of RTL models, the co-existence of RTL events with network events has to be addressed. RTL events model the setup of logical values on ports and signals and they have an instantaneous propagation, i.e., they are triggered at the same simulation time in which the corresponding values are changing. Furthermore, except for tri-state logic, ports and signals have always a value associated to them, i.e., sentences like “nothing is on the port” are meaningless. Instead, network events are mainly related to the transmission of packets; each transmission is not instantaneous because of transmission delay, during idle periods the channel is empty, and the repeated transmission of the same packet is possible and leads to distinct network events.

In SCNSL, RTL task models handle packet-level events by using three ports signaling the start and the end of each packet transmission, and the reception of a new packet, respectively. To each task instance are associated one or more taskproxies. A taskproxy is an object able to translate network events into RTL events and viceversa. In particular, each RTL task has to write on a specific port when it starts the transmission of a packet while another port is written by the corresponding taskproxy when the transmission of the packet is completed. A third port is used to notify the task about the arrival of a new packet. With this approach each transmission/reception of a packet is detected even if packets are equal.

The last issue regards the handling of packets of different size. Real world protocols use packets of different sizes while RTL ports must have a constant width set at compile-time. SCNSL solves this problem by creating packet ports with the maximum packet size allowed in the network scenario and by using an integer port to communicate the actual packet size. In this way a taskproxy or a receiver task can read only the actual used bytes, thus obtaining a correct simulation.

10.3.4 The notion of packet

One key object inside a network simulator is the representation of a packet. In SCNSL such a packet is described by the `Packet_t` class. This class is an abstraction of an actual packet, and in fact its main fields are a buffer of bytes, storing the actual transmitted data, and an integer describing the size of the buffer. This allows to use the `Packet_t` class to model also variable-length packets or packets affering to different protocols. Such a Packet is required to standardize the data exchanged between the SCNSL components, and thus, increasing their modularity and reusability.

While RTL types are not enough flexible to implement such a standard packet, the TLM payload could be a valid choice. Unfortunately, the TLM payload has been designed to describe intra-node communication, like memory accesses. Thus, it contains many fields which would be unused, leading to a waste of simulation resources. Moreover, there is not need for adherence of the packet implementation to the TLM standard, since the `Packet_t` is a SCNSL internal class completely hidden to users.

10.3.5 RTL & TLM models coexistence

The SystemC allows users to design modules at different abstraction levels, i.e. at RTL and TLM. These modules can be mixed inside a simulation instance by using special modules to interconnect them. Such modules are called *transactors* and their role is to translate values read from RTL ports and signals into TLM payloads, and viceversa. This idea has been applied also to the design of SCNSL, in order to allow the design of tasks different abstraction levels. This issue has been addressed in two steps:

1. Translation of RTL and TLM exchanged data into an abstract and independent internal format.
2. Standardization of network communication mechanisms, in order to avoid to implement a new transactor for each module. Only two transactors are required: between RTL and the SCNSL internal format, and between TLM and SCNSL internal format.

Step 1 has been resolved by using SCNSL internal `Packet_t` class, as described in Section 10.3.4.

Step 2 has lead to the creation of network communication transactors, which have been named *TaskProxies*. Each TaskProxy can be then considered as the class which separates the user space, from the network simulation backend implemented by SCNSL.

Moreover, the combined use of taskproxies and internal packets have the advantage that SCNSL internal types shall be unrelated to SystemC types. Thus, SCNSL internal classes have been implemented mixing SystemC and C++ code to optimize simulation performances.

10.3.6 Simulation planning

In several network simulators, special events can be scheduled during the setup of the scenario, for instance node movements, link status changes, and traffic activation. This feature is important because it allows to simulate the model into a dynamic network context. In SCNSL the simulation kernel has not its own event dispatcher, hence this feature has been implemented into an optional class, called `EventsQueue_t`. Even if SystemC

allows to write in each task the code which triggers such events, the choice of managing them in a specific class of the simulator leads to the following advantages:

- **Standard API**: the events queue provides a clear interface to schedule a network event without directly interacting with the Network class or altering node implementation.
- **Simplified user code**: Network events are more complex then System ones; the events queue hides such complexity thus simplifying user code and avoiding setup errors.
- **Higher performance**: the management of all the events inside a single class improves performance; in fact the events queue is built around a single SystemC thread, minimizing memory usage and context switching.

This class can be used also to trigger new events defined through user-defined functions. The only constraint is that such functions shall not block the caller, i.e., the event queue, to allow a correct scheduling of the following events.

10.3.7 Implementation of network protocols and channels

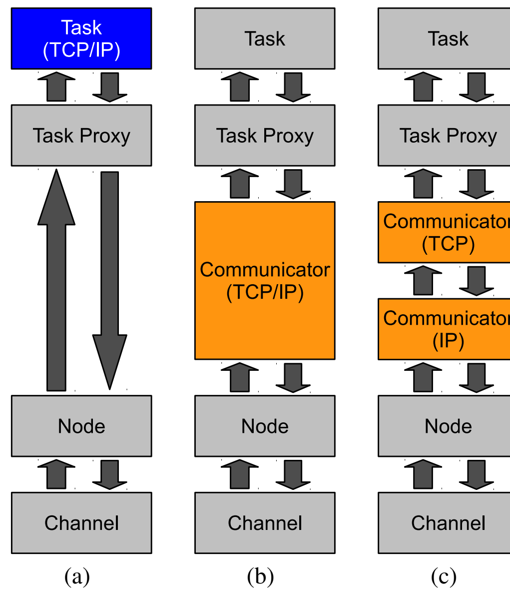


Fig. 10.5. Examples of possible Communicator chains, implementing a TCP/IP-like transmission: (a) Empty communicator chain; the protocol is implemented by the user. (b) A single communicator module, implementing all the protocol layers. (c) A long communicator chain with an element for each protocol layer.

Communication protocols are a fundamental component for network simulators. Each scenario could require a different protocol, which provides different API's, while simulator components require a standard API to allow code reuse. These opposite requirements can be satisfied as follow:

- Each task will interact with the simulator by using standard API's. Such API's will allow to perform only the basic and general communication operations, i.e. carrier sensing and packets exchanging. Advanced functionalities can be implemented by encapsulating additional information inside packets. In SCNSL, the API's are implemented by the *TaskProxies*, as described in Section 10.3.5.
- In the backend, a standard packet format will be used, to allow component reuse, as described in Section 10.3.4.
- In the backend, components will implement an interface, to standardize the exchanging of packets and carrier status. In SCNSL this interface has been named *Communicator*. By using the Communicator interface, backend components can be joint into chains, to create complex behaviors.

In SCNSL, there are two options to use a protocol:

1. The user does not adopt any provided protocol model, since its implementation is the design goal. Thus, the protocol will be modeled as any other functionality, at task level (Figure 10.5(a)). For instance, ad-hoc protocols and protocols that will be synthesized in hardware fall in this case.
2. The user relies on protocol models provided by the simulator. A complete protocol stack can be implemented by using a single simulator component (Figure 10.5(b)), or by using a component for each protocol layer (Figure 10.5(c)). Using a single component simplify the scenario creation, since fewer modules shall be instantiated, but using many components allows code reuse (e.g. TCP/IP and UDP/IP stacks have the IP layer in common). SCNSL supports both these implementation strategies.

The use of communicators has also the advantage to allow to write components to change communication behavior, e.g. queues to buffer packets under transmission. The only constraint about communicators chains is that each chain shall have at one end a taskproxy, and on the other end a node.

Transmission media have been modeled in components named *Channels*. In SCNSL, there are many channel models, according with the medium type and the physical simulation accuracy. For instance there are unidirectional links, full-duplex links, half-duplex links and wireless channels. Each channel type implements the `Channel_if_t` interface, in order to allow to change the channel model, without changing node implementation. Channels are the most complex objects, because they manage transmissions and, for this reason, they must be highly optimized. For instance, in the wireless model, the channel transmits packets and simulates their transmission delay; it must take into account node movements, check which nodes are able to receive a packet, and verify if a received packet has been corrupted due to collisions. The standard SystemC kernel does not address these aspects directly, but it provides important primitives such as concurrency models and events. The channel class uses these SystemC primitives to reproduce transmission behavior. In particular, it is worth to note that SCNSL does not have its own scheduler since it exploits the SystemC scheduler by mapping network events on standard SystemC events.

10.4 Experimental results

The SystemC Network Simulation Library has been used to model a wireless sensor network application consisting of a master task which repeatedly polls sensor tasks

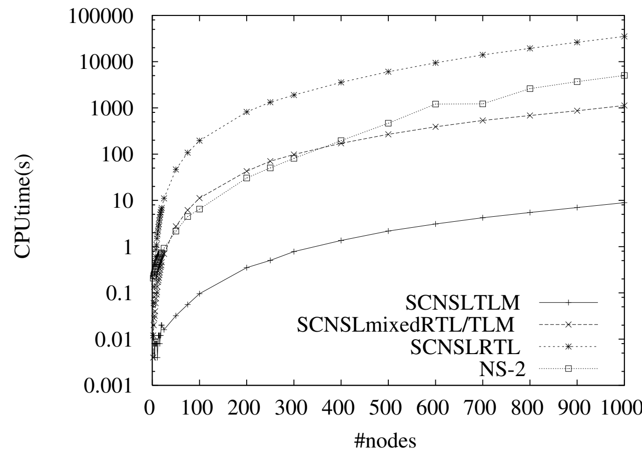


Fig. 10.6. CPU time as a function of the number of simulated nodes for the different tested tools and node abstraction levels.

to obtain data. The chosen communication protocol model reproduces a subset of the IEEE 802.15.4 standard, i.e., peer un-slotted transmissions with acknowledge [111].

Different scenarios have been simulated with SCNSL by using tasks at different abstraction levels: *1)* all nodes at TLM level, *2)* all nodes at RTL, and *3)* master task at RTL and sensor tasks at TLM. The designer had written 77 code lines for the `sc_main()`, 633 code lines for the RTL task and 204 code lines for the TLM task.

Figure 10.6 shows the CPU time as a function of the number of nodes for the three scenarios and for a simulation provided by NS-2 representing the behavior of a pure network simulator. A logarithmic scale has been used to better show results. Simulations have been performed on the Intel Xeon 2.8 MHz with 8 GB of RAM and 2.6.23 Linux kernel; CPU time has been computed with the `time` command by summing up user and system time.

The speed of SCNSL simulations at TLM level is about two-order-magnitude higher than in case of NS-2 simulation showing the validity of SCNSL as a tool for efficient network simulation. Simulations at RT level are clearly slower because each task is implemented as a clocked finite state machine as commonly done to increase model accuracy in System design. However a good trade-off between simulation speed and accuracy can be achieved by mixing tasks at different abstraction levels; in this case, experimental results report about the same performance of NS-2 with the advantage that at least one task is described at RT level.

10.5 Conclusions

This work has presented a SystemC-based approach to model and simulate networked embedded systems. As a result, a single tool has been created to model both the embedded systems under design and the surrounding network environment. Different issues have been solved to reconcile System design and Network simulation requirements while preserving execution efficiency. In particular, the combined simulation of RTL system

models and packet-based networks has been faced. Experimental results for a large network scenario show nearly two-order-magnitude speed up with respect to NS-2 with TLM modeling and about the same performance as NS-2 with a mixed TLM/RTL scenario.

The work described in this chapter appears in the following publications: [73, 77].

11

A final case study

This chapter presents a more complex case study w.r.t. the ones presented in previous chapters. The application design starts from high level specification, down to an actual board. Since the application relies also on the presence of a middleware, at NWV the integrated CASSE-AME methodology has been applied.

The purpose of this chapter is twofold:

1. It constitutes a complex example of methodology application, useful to further clarify the main concepts.
2. It allows to verify the accuracy of proposed methodologies and tools.

The next sections are organized as follows. The scenario is depicted in Section 11.1. The modeling at SYSV in AME and the choice of the middleware are reported in Section 11.2 and Section 11.3 respectively. Section 11.4 presents the NWV refinement, while design steps at PLV are summarized in Section 11.5. The methodology validation is reported in Section 11.6. Final remarks are drawn in Section 11.7.

11.1 Case study description

The communication-aware design methodology described in Figure 5.2 has been applied to an application for remotely-assisted training. Each user wears a 3D accelerometer whose data are used to compute the step rate and the run speed which are then forwarded to the trainer's display.

11.2 System View modeling

The first design step is to model the application in SystemC. Application is decomposed into its functional tasks. A SystemC block representing the middleware is also present and tasks communicate each other through its services. In this step, the network is not modeled to speed-up simulation (System View). In the case study an object-oriented middleware has been modeled since the considered actual implementations follow this paradigm. However the methodology can be applied to different paradigms available in literature, e.g., publish-subscribe, tuple-space and database paradigms.

The application is decomposed into three types of tasks:

1. the Sensor task (S) senses the acceleration values;
2. the Processing task (P) computes step rate and run speed;
3. the Display task (D) displays computed data.

There is a sensor and a processing task for each user while there is only one display task for all the users. Acceleration sampling frequency is 7 Hz.

To model the application according to CASSE framework, communication requirements are extracted directly from the specification as follows. Each sensor task communicates with its corresponding processing task which communicates with the unique display task. Each S task produces 7 samples per second each represented on a 4-bytes float number for a total requested throughput of 672 b/s. With the same frequency, each P task generates a message with the step rate and the run speed, described as two 4-bytes float numbers; furthermore each message contains a 12 bytes header used by the D task to identify the P task; the total requested throughput is 1120 b/s.

11.3 Choosing the middleware

The next step is the choice of the appropriate middleware to be introduced in CASSE specification. A library of models of actual middleware implementations has been created following the methodology described in Section 5.2.2. Middleware characteristics are reported in Table 11.1.

Name	m (byte)	q (byte)	Ack (byte)	Method encoding (byte)	Setup Packets	Communication schema ORB	Communication schema Service	Binary size
ZigBee OOM	1	16	16	0	6	centralized	distributed	122.2 KB
ZeroC Ice-E	1	27	27	2	12	centralized	distributed	4.1 MB
ZeroC Ice	1	27	27	2	12	centralized	distributed	45.3 MB
Java RMI	1	32	22	1	16	centralized	distributed	77.4 MB
Java RMI/IIOP	1	111	34	1	29	centralized	distributed	77.4 MB

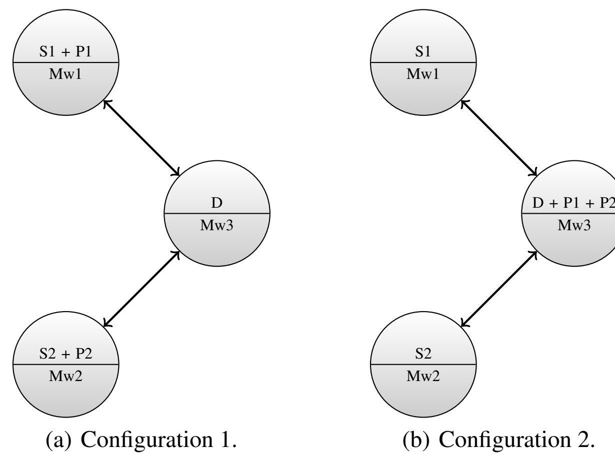
Table 11.1. Model library for actual middleware implementations.

Java RMI/IIOP is not suitable for embedded systems, due to its high memory usage and communication overhead. Java RMI and ZeroC Ice are similar, but the latter requires less memory, especially in its version for embedded systems, namely Ice-E. The ZigBee OOM consists of a standard ZigBee protocol stack with an additional layer to reproduce basic object-oriented services. The method encoding overhead is set to zero since the method is encoded by using an integer which is already accounted inside q and not by using a specific field as in the other middleware implementations. Zero-C Ice has the highest overhead for method encoding.

ZigBee OOM has been chosen as target middleware in this case study for its low memory footprint. The communication requirements after the introduction of the ZigBee OOM can be computed with the parameters contained in the Table. In particular, the throughput requested by application tasks with the middleware overhead is $7Hz * (m * 12B + q + \text{Ack} + \text{MethodEncoding} * \text{methodNameLenght}) = 2464b/s$ for S tasks and $7Hz*(m*(8B+12B)+q+\text{Ack}+\text{MethodEncoding}*\text{methodNameLenght}) = 2912b/s$ for P tasks.

	CASSE with Middleware		Actual Behavior		Error %	
	No. of Packets	Bytes	No. of Packets	Bytes	No. of Packets	Bytes
ZigBee OOM cfg 1	20008	520072	20012	260190	0.02	0.06
ZigBee OOM cfg 2	20008	440072	20012	440380	0.02	0.07
Zero-C Ice cfg 1	20014	920000	20024	920260	0.05	0.03
Zero-C Ice cfg 2	20014	840000	20024	840260	0.05	0.03
Java RMI cfg 1	20022	830000	20253	834327	1.14	0.52
Java RMI cfg 2	20022	750000	20237	753081	1.06	0.41
Java RMI/IIOP cfg 1	20048	1740000	20109	1746595	0.3	0.38
Java RMI/IIOP cfg 2	20048	1660000	20109	1666595	0.3	0.4

Table 11.2. Experimental results.



(a) Configuration 1. (b) Configuration 2.

Fig. 11.1. Task configurations considered in the case study.

11.4 CASSE modeling and network synthesis

At this point of the design flow, the problem has been formulated according to CASSE framework. For space limits, formalization details are not shown here, but only the key ideas are reported. Assuming that the main design goal is the reduction of the economic cost, the CASSE methodology forces the instantiation of just two kind of nodes, i.e., one to run the D task, and one to run the S task (clearly, there will be an instance of this node for each user). P tasks can be hosted either by the sensing nodes or all together by the display node. The resulting possible configurations are reported in Figure 11.1. By using the CASSE analytical model, we estimated the total number of bytes transmitted by a system of two users and one trainer in both configurations; values at middleware level are reported in the first column of Table 11.2. Since the economic cost of the two solutions is equal, these results lead to choose *Configuration 2* which minimizes the number of transmitted bytes.

11.5 Remaining design steps

Finally, per-node HW/SW refinement and partitioning is performed. For instance, an actual SystemC RTL model of the accelerometer can be introduced and the sensor task can be modified to interact with the device driver of the accelerometer. These design details are not shown, since they are out of the scope of the work.

11.6 Methodology validation

To validate the proposed methodology, both candidate scenarios have been deployed on boards and tested by using the corresponding actual implementations of middleware. For ZigBee, a development kit by Texas Instruments has been used while for the other middleware types a stardard TCP/IP embedded platform has been used. The scenario has involved two users and one trainer, for a total of three boards. The execution time is the same as for the results obtained with CASSE. The total number of transmitted packets and transmitted bytes at middleware level is reported in the second column of Table 11.2 for both configurations.

The table allows the comparison of actual results with those derived by using the CASSE methodology; the error is always less than 1.2% in all cases showing that:

1. the methodology to characterize middleware implementations is correct, since their introduced overhead has been estimated very accurately;
2. the formal specification provided by CASSE leads to results which well represent the behavior of the actual implementation;

11.7 Conclusions

This case study is a complex example used to clarify how the proposed design flow can be successfully applied to real life scenarios. The application design flow has been applied starting from specification, down to actual boards. Reported experimental results have shown a very high accuracy of CASSE modeling and AME/SCNSL simulation, thus, validating the global flow. Moreover, reported results allow to validate also the middleware characterization methodology described in Section 5.2.

12

Conclusions

The design of Networked Embedded Systems is a challenging research field, which poses new unresolved problems (Chapter 1). The main difference w.r.t. traditional design flows is related to the communication aspects, which play a central role in the behavior and can be considered a design-space dimension when the network is not fixed a priori. Whilst traditional application behavior is specified on per-node basis, NES's behavior is specified in terms of global properties and node interactions, thus requiring to shift the focus of design and verification from each single node to the whole distributed application.

Exploiting these two key ideas, the thesis proposed a general design flow (Chapter 3) which starting from high level specifications leads to the actual system. The objective is to try to overcome the limits of traditional methodologies, and, at the same time, to allow their re-use during the last design phase (i.e. at PLV). The three main design macro-phases (SYSV, NWV & PLV) are supported by specific sub-methodologies and tools, developed during these years of Ph.D. studies.

The core methodology of the thesis is CASSE (Chapter 4), which allows the refinement of communication-related components at NWV. Other important topics, which have a great practical applicability, have been addressed, such as the middleware integration (Chapter 5) and the dependability assessment (Chapters 6 7 & 8).

The applicability and validity of proposed methodologies is supported by results reported in the case studies (in particular, Chapter 11) and by the implementation of actual tools (Chapters 9 & 10).

Since the thesis addresses many different topics, a lot of future work is possible, such as:

- Extending CASSE to better modeling mobility.
- Introducing some probabilistic behavior in the CASSE formal model, to allow more complex considerations about the dependability.
- Improving the SCNSL implementation, checking if it could be adopted to model also Network on Chips.
- Investigating deeper integrations between SCNSL and standard SystemC libraries, such as the Verification and the Analog Mixed Signal ones.
- Improving and optimizing all developed tools.

A

Publications

List of publications made during these years of Ph.D. studies, divided by type and in reverse chronological order.

A.1 Articles published on journals

1. N. Bombieri, M. Ferrari, F. Fummi, G. Di Guglielmo, G. Pravadelli, F. Stefanni, A. Venturelli.
   HIFSuite: Tools for HDL Code Conversion and Manipulation.
   EURASIP Journal on Advances in Signal Processing, 2010.

A.2 Papers published in conference proceedings

1. F. Fummi, D. Quaglia, F. Stefanni.
   Communication-aware Design Flow for Dependable Networked Embedded Systems.
   IEEE International Symposium on Circuits and Systems (ISCAS 2011).

2. N. Bombieri, G. Di Guglielmo, L. Di Guglielmo, M. Ferrari, F. Fummi, G. Pravadelli, F. Stefanni, A. Venturelli.
   HIFSuite: Tools for HDL Code Conversion and Manipulation.
   IEEE International High Level Design Validation and Test Workshop (HLDVT'10).

3. F. Fummi, G. Lovato, D. Quaglia, F. Stefanni.
   Modeling of Communication Infrastructure for Design-Space Exploration.
   IEEE Forum on specification and Design Languages (FDL'10).

4. F. Fummi, D. Quaglia, F. Stefanni.
   Time-varying Network Fault Model for the design of dependable networked embedded systems.
   IEEE Euromicro Conference on Digital System Design (DSD'09).

5. G. Di Guglielmo, F. Fummi, M. Hampton, G. Pravadelli, F. Stefanni.
   The role of parallel simulation in functional verification.
   IEEE International High Level Design Validation and Test Workshop (HLDVT'08).

6. F. Fummi, D. Quaglia, F. Stefanni.
   Network fault model for dependability assessment of networked embedded systems.
   IEEE International Symposium Defect and Fault Tolerance of VLSI Systems (DFTVS'08).

7. F. Fummi, D. Quaglia, F. Stefanni.
   A SystemC-based framework for modeling and simulation of networked embedded systems.
   IEEE Forum on specification and Design Languages (FDL'08).

A.3 Other publications

1. F. Fummi, D. Quaglia, F. Stefanni.
   SystemC Simulation of Networked Embedded Systems.
   Languages for Embedded Systems and their Applications.
   Springer Science, (2009). pp.201- 211 ISBN:978-1-4020-9713-3.
   Book chapter.

2. G. Di Guglielmo, F. Fummi, G. Pravadelli, F. Stefanni.
   HIFSuite: tools for HDL code manipulation.
   IEEE/ACM Design, Automation and Test in Europe (DATE'07).
   University booth.

B

Acknowledgments

In English

Some years have passed since when I have started my Ph.D. studies, and, during such a time, a lot of things have happened in both my public and personal life. Nice things, like Ph.D. schools, abroad period and a new house, but also very hard times. Thus, coming to the end of this part of my life, I like to say thank you to many people who have been by my side in this “journey”.

First of all, I have to remember all my colleagues, for the support and for the friendly academic surrounding. Especially, I like to remember Prof. Franco Fummi and Researcher Davide Quaglia, to have given me the opportunity to do the doctorate under their leadership, and to have helped me to acquire new professional skills.

Outside the University, I have the luck to be loved by many people without whom probably I would not be here today. I do not want to mention them here, but I just like to let them know that I am pleased to have met them in my life, and that I hope they will continue to be part of for a long time.

In Italian

Sono passati alcuni anni da quando ho iniziato il corso di dottorato, e, durante tale periodo, sono successe molte cose sia nella mia vita pubblica, sia in quella privata. Cose piacevoli, quali le scuole di dottorato, un periodo all’estero e una nuova casa, ma anche situazioni difficili. Pertanto, giungendo al termine di questa parte della mia vita, voglio ringraziare le molte persone che sono state al mio fianco in questo “viaggio”.

Innanzi tutto, devo ricordare i miei colleghi, per il supporto datomi e per l’amichevole ambiente di lavoro. In modo particolare, ringrazio il Prof. Franco Fummi e il Ricercatore Davide Quaglia, per avermi dato l’opportunità di svolgere il dottorato di ricerca sotto la loro guida, e per avermi aiutato ad acquisire nuove competenze professionali.

Al di fuori dell’Università, ho la fortuna di avere molte persone che mi vogliono bene, senza le quali probabilmente oggi non sarei qui. Non voglio menzionarle ora, ma desidero sappiano che sono contento di averle incontrate nella mia vita, e che spero continuino a farne parte per molto tempo.

C

Sommario (in Italian)

Oggigiorno i Sistemi Dedicati di Rete (Networked Embeddedd Systems – NES) sono una tecnologia pervasiva. Il loro utilizzo comprende applicazioni per il monitoraggio, per l'automazione delle case, e per compiti in ambienti critici. La loro crescente complessità richiede nuove metodologie per poter effettuare efficientemente le fasi di progettazione e verifica. Questo lavoro presenta un flusso di sviluppo generico per NES, supportato dall'implementazione di programmi per la loro applicazione. Il flusso di sviluppo sfrutta il linguaggio SystemC, e considera la rete come una dimensione dello spazio di progetto. La metodologia di base comprende estensioni per considerare anche i casi in cui il NES sia implementato usando un middleware o in cui siano presenti dei requisiti di affidabilità. Inoltre, sono stati implementati dei programmmi di traduzione per consentire l'adozione della metodologia proposta con design scritti in altri linguaggi per la descrizione dello hardware.

References

1. A. A. S. Danthine. Protocol representation with finite-state models. *IEEE Trans. on Communications and Parallel Distrib. Syst.*, 28(4):632–643, Apr. 1980.
2. J. A. Abraham and K. Fuchs. Fault and Error Models for VLSI. *Proc. of the IEEE*, 74(5):639–654, May 1986.
3. M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital systems testing and testable design*. IEEE Press, 1992.
4. Abdelnaser Adas. Traffic models in broadband networks. *IEEE Communications Magazine*, pages 82–89, 1997.
5. Advanced Micro Devices. *Advanced micro devices. 1978. The AM2910, a complete 12-bit microprogram sequence controller.* In AMD Data Book, AMD Inc.
6. A. Agarwal, C. Iskander, H.T. Multisystems, and R. Shankar. Survey of network on chip (noc) architectures & contributions. In *Journal of Engineering, Computing and Architecture*, 2009.
7. N. Agliada, A. Fin, F. Fummi, M. Martignano, and G. Pravadelli. On the Reuse of VHDL Modules into SystemC Designs. In *Proc. of IEEE FDL*, 2001.
8. N. A. Ahmed, I. A. Aref, F. Rodriguez-Salazar, and K. Elgaid. Wireless channel model based on soc design methodology. In *Proceedings of the 2009 Fourth International Conference on Systems and Networks Communications*, ICSNC '09, pages 72–75, Washington, DC, USA, 2009. IEEE Computer Society.
9. Ian F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks Elsevier Journal*, 38(4):393–422, March 2002.
10. ALDEC. DVM. `http://www.aldec.com/products/`.
11. Per Andersson and Martin Host. Uml and systemc – a comparison and mapping rules for automatic code generation. In Eugenio Villar, editor, *Embedded Systems Specification and Design Languages*, volume 10 of *Lecture Notes in Electrical Engineering*, pages 199–209. Springer Netherlands, 2008.
12. D. B. Armstrong. A deductive method for simulating faults in logic circuits. In *IEEE Trans. Comput.*, volume C-21, no. 5, pages 464–471, May 1972.
13. A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. *Proc. 3rd Information Survivability Workshop*, pages 7–12, 2000.
14. AWE Communications. WinProp: Software-Tool for the Planning of Mobile Communication Networks. URL: http://www.awe-communications.com.
15. Amol Bakshi and Viktor K. Prasanna. Algorithm design and synthesis for wireless sensor networks. In *International Conference on Parallel Processing (ICPP 2004)*, pages 15–18 (1). IEEE Society, August 2004.
16. Felice Balarin and Roberto Passerone. Specification, Synthesis and Simulation of Transactor Processes. *IEEE Trans. on CAD*, accepted for future publication.

17. Twan Basten, Marc Geilen, and Harmke de Groot. *Ambient Intelligence: Impact on Embedded System Design*. Kluwer Academic Publishers, 2003.
18. N. K. Bhatti and R. D. Blanton. Diagnostic Test Generation for Arbitrary Faults. In *Proc. of IEEE ITC*, pages 1–9, 2006.
19. Tobias Bjerregaard and Shankar Mahadevan. *A Survey of Research and Practices of Network-on-Chip*.
20. T. Blank. A survey of hardware accelerators used in computer aided design. *IEEE Design & Test Comput.*, Aug. 1984.
21. Massimo Bombana and Francesco Bruschi. SystemC-VHDL Co-Simulation and Synthesis in the HW Domain. In *Proc. of ACM/IEEE DATE*, pages 106–111, 2003.
22. N. Bombieri, G. Di Guglielmo, L. Di Guglielmo, M. Ferrari, F. Fummi, G. Pravadelli, F. Stefanni, and A. Venturelli. Hifsuite: Tools for hdl code conversion and manipulation. In *High Level Design Validation and Test Workshop (HLDVT), 2010 IEEE International*, pages 40 –41, jun. 2010.
23. N. Bombieri, M. Ferrari, F. Fummi, G. Di Guglielmo, G. Pravadelli, F. Stefanni, and A. Venturelli. HIFSuite: Tools for hdl code conversion and manipulation. *EURASIP JOURNAL ON ADVANCES IN SIGNAL PROCESSING*, pages 1–20, 2010.
24. N. Bombieri, F. Fummi, and G. Pravadelli. A Mutation Model for the SystemC TLM 2.0 Communication Interfaces. In *Proc. of ACM/IEEE DATE*, pages 396–401, 2008.
25. Nicola Bombieri, Franco Fummi, and Graziano Pravadelli. A Methodology for Abstracting RTL Designs into TL Descriptions. In *Proc. of ACM/IEEE MEMOCODE*, pages 103–112, 2006.
26. Nicola Bombieri, Franco Fummi, and Graziano Pravadelli. Towards Equivalence Checking Between TLM and RTL Models. In *Proc. of ACM/IEEE MEMOCODE*, pages 113–122, 2007.
27. A. Bonivento, L. P. Carloni, and A. Sangiovanni-Vincentelli. Platform-based design of wireless sensor networks for industrial applications. In *Proc. of Design, Automation and Test in Europe (DATE'06)*, pages 1–6 (1), Munich, March 2006. IEEE Society.
28. D.S. Brahme, S. Cox, J. Gallo, M. Glasser, W. Grundmann, C. Norris Ip, W. Paulsen, J.L. Pierce, J. Rose, D. Shea, and K. Whiting. The Transaction-Based Verification Methodology. Technical Report CDNL-TR-2000-0825, Cadence Berkeley Labs, 2000.
29. Herman Bruyninckx and Ren van de Molengraft. *Embedded Control System Design*. Wikibooks, 2010.
30. C. Zhu et al. A comparison of active queue management algorithms using the OPNET modeler. *IEEE Communications Magazine*, 40(6):158–167, Jun. 2002.
31. A. Castelnuovo, A. Fedeli, A. Fin, F. Fummi, G. Pravadelli, U. Rossi, F. Sforza, and F. Toto. A 1000X Speed Up for Properties Completeness Evaluation. In *Proc. of IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 18–22, 27-29 Octorber 2002.
32. Center for Electronic Systems Design. Metropolis. URL: http://embedded.eecs.berkeley.edu/metropolis/index.html.
33. Center for Embedded and Computer Systems. Specc. URL: http://cecs.uci.edu/˜specc/.
34. Center for Hybrid and Embedded Software System. Ptolemy. URL: http://ptolemy.eecs.berkeley.edu/index.htm.
35. M. Ceriotti, L. Mottola, G. P. Picco, A. L. Murphy, S. Guna, M. Corrnta, and P. Zanon. Teeny lime. URL: http://teenylime.sourceforge.net.
36. Certess. Certitude. `www.certess.com/product/Certitude_datasheet.pdf`.
37. R. Chae-Eun, J. Han-You, and Ha Soonhoi. Many-to-many core-switch mapping in 2-d mesh noc architectures. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2004.
38. K.-T. Cheng and J.-Y. Jou. A single-state-transition fault model for sequential machines. *Computer-Aided Design*, pages 226–229, Nov. 1990.

39. W. T. Cheng and M. L. Yu. Differential fault simulation - A fast method using minimal memory. In *Proc. Design Automation Conf.*, pages 424–428, June 1989.
40. Alexandre Chureau, Yvon Savaria, and El Mostapha Aboulhamid. The Role of Model-Level Transactors and UML in Functional Prototyping of Systems-on-Chip: A Software-Radio Application. In *Proc. of ACM/IEEE DATE*, pages 698–703, 2005.
41. International Electrotechnical Commission. Functional safety of electrical/electronic/programmable electronic safety-related systems. http://www.iec.ch/zone/fsafety/.
42. M. Conti and D. Moretti. System level analysis of the bluetooth standard. In *Proc. IEEE DATE*, volume 3, pages 118–123, Mar. 2005.
43. Controller Area Network. URL: http://www.iso.org/.
44. C. Cote and Z. Zilic. Automated SystemC to VHDL Translation in Hardware/Software Codesign. In *Proc. of IEEE ICECS*, pages 717–720, 2002.
45. D. Bertozzi et al. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Trans. Parallel Distrib. Syst.*, 16(2):113–129, Feb. 2005.
46. D. Desmet et al. Timed executable system specification of an ADSL modem using a C++ based design environment: a case study. In *Proc. IEEE CODES*, pages 38–42, 1999.
47. SystemC Network Simulation Library – version Beta, 2008. URL: http://sourceforge.net/projects/scnsl.
48. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11:34–41, April 1978.
49. G. Di Guglielmo, F. Fummi, M. Hampton, G. Pravadelli, and F. Stefanni. The role of parallel simulation in functional verification. In *High Level Design Validation and Test Workshop, 2008. HLDVT '08. IEEE International*, pages 117 –124, nov. 2008.
50. G. Di Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli. Improving high-level and gate-level testing with fate: a functional atpg traversing unstabilized efsms. *Computers & Digital Techniques, IET*, 1:187–196, May 2007.
51. D. Dietterle, J. Ebert, G. Wagenknecht, and R. Kraemer. A wireless communication platform for long-term health monitoring. In *Proc. IEEE International Conference on Pervasive Computing and Communications Workshop*, Mar. 2006.
52. Thomas Donald and Moorby Phillip. *The Verilog Hardware Description Language*. Springer Science, 2002.
53. Edwin Naroska. FreeHDL. `http://www.freehdl.seul.org/`.
54. Peter Van Eijk and Jeroen Schot. An exercise in protocol synthesis. In *Formal Description Techniques IV. North-Holland*, pages 117–131, 1991.
55. M. El-Darieby and A. Bieszczad. Intelligent mobile agents towards network fault management automation. In *Proc. of the Sixth IFIP/IEEE International Symposium on Integrated Network Management*, pages 611–622, 1999.
56. Denise W. Gurer et al. An artificial intelligence approach to network fault management.
57. G. Jacques-Silva et al. A network-level distributed fault injector for experimental validation of dependable distributed systems. In *Proc. of International Computer Software and Applications Conference (COMPSAC)*, 2006.
58. European Commission. Hybrid control: Taming heterogeneity and complexity of networked embedded systems - HYCON. *FP6-IST-511368-NOE. URL: http://www.ist-hycon.org*, 2004.
59. Andrea Fedeli, Franco Fummi, and Graziano Pravadelli. Properties Incompleteness Evaluation by Functional Verification. *IEEE Trans. Comput.*, 56(4):528–544, 2007.
60. F. Ferrandi, G. Ferrara, S.Sciuto, A. Fin, and F. Fummi. Functional test generation for behaviorally sequential models. In *Design, Automation, and Test in Europe*, pages 403–410, 2001.
61. F. Ferrandi, F. Fummi, L. Gerli, , and D. Sciuto. Symbolic functional vector generation for VHDL specifications. In *IEEE DATE*, pages 226–446, 1999.
62. FFA. Do-178b. http://www.do178site.com/.

63. F.Ferrandi, F.Fummi, and D.Sciuto. Implicit test generation for behavioral VHDL models. In *Proc. IEEE Int. Test Conf. (ITC)*, pages 587–596, Oct. 1998.
64. Fieldbus Foundation. URL: http://www.fieldbus.org.
65. J. Fleischmann and K. Buchenrieder. Prototyping networked embedded systems. *IEEE Computer*, 32(2):116–119, Feb. 1999.
66. F. Fummi, P. Gallo, S. Martini, G. Perbellini, M. Poncino, and F. Ricciato. A timing-accurate modeling and simulation environment for networked embedded systems. In *Proc. ACM/IEEE Design and Automation Conf. (DAC)*, pages 42–47, Jun. 2003.
67. F. Fummi, C. Marconcini, and G. Pravadelli. Logic-level mapping of high-level faults. *INTEGRATION, the VLSI Journal*, 38:467–490, 2004.
68. F. Fummi, G. Perbellini, R. Pietrangeli, and D. Quaglia. A middleware-centric design flow for networked embedded systems. In *Proc. IEEE Conf. on Design, Automation and Test in Europe (DATE)*, Apr. 2007.
69. F. Fummi, G. Perbellini, D. Quaglia, and R. Trenti. Exploration of network alternatives for middleware-centric embedded system design. In *Proc. of Euromicro Conf. on Digital System Design (DSD)*, 2010.
70. F. Fummi, G. Perbellini, D. Quaglia, and S. Vinco. AME: an abstract middleware environment for validating networked embedded systems applications. In *HLDVT '07: Proceedings of the 2007 IEEE International High Level Design Validation and Test Workshop*, pages 187–194, Washington, DC, USA, 2007. IEEE Computer Society.
71. F. Fummi, D. Quaglia, and F. Stefanni. Communication-aware middleware-based design-space esploration for networked embedded systems.
72. F. Fummi, D. Quaglia, and F. Stefanni. Network fault model for dependability assessment of networked embedded systems. In *Proc. of IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'08)*, pages 54–62, 2008.
73. F. Fummi, D. Quaglia, and F. Stefanni. Systemc simulation of networked embedded systems. In *Languages for Embedded Systems and their Applications*, volume 36, pages 201–211. Springer Science, 2009.
74. F. Fummi, D. Quaglia, and F. Stefanni. Time-varying network fault model for the design of dependable networked embedded systems. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 225 –228, aug. 2009.
75. F. Fummi, D. Quaglia, and F. Stefanni. Modeling of communication infrastructure for design-space exploration. In *Proc. of IEEE Forum on specification & Design Languages (FDL)*, 2010.
76. F. Fummi, D. Quaglia, and F. Stefanni. Communication-aware design flow for dependable networked embedded systems. In *Proc. of IEEE International Symposium on Circuits and Systems (ISCAS)*, 2011.
77. Franco Fummi, Davide Quaglia, and Francesco Stefanni. A systemc-based framework for modeling and simulation of networked embedded systems. In *Forum on Specification, Verification and Design Languages (FDL'08)*, pages 49–54, Stuttgart, Germany, September 2008. IEEE Computer Socety.
78. R. D. Gardner and D. A. Harle. Alarm correlation and network fault resolution using the kohonenself-organising map. In *Proc. of IEEE Global Communication Conference*, pages 1398–1402, 1997.
79. Gerald Combs et al. Wireshark. URL: http://www.wireshark.org.
80. Gnu Operating System. Bison. `http://www.gnu.org/software/bison/`.
81. H. C. Godoy and R. E. Vogelsberg. Single Pass Error Effect Determination (SPEED). *IBM Technical Disclosure Bulletin*, 13:3344–3443, April 1971.
82. G. Gogniat, M. Auguin, L. Bianco, and A. Pegatoquet. Communication synthesis and hw/sw integration for embedded system design. In *Hardware/Software Codesign, 1998. (CODES/-CASHE '98) Proceedings of the Sixth International Workshop on*, pages 49 –53, March 1998.

83. S. Gosh and T. J. Chakraborty. On Behavior Fault Modeling for Digital Design. *Journal of Electronic Testing: Theory and Applications (JETTA)*, 2(2):31–42, June 1991.
84. Mohamed G. Gouda. Protocol verification made simple: a tutorial. *Computer Networks and ISDN Systems*, 25(9):969–980, 1993.
85. Green Mountain Computing Systems. Hc11. http://www.gmvhdl.com/hc11core.html.
86. GreenSocs. KaSCPar. `http://www.greensocs.com/en/projects/KaSCPar`.
87. Abdelaziz Guerrouat and Harald Richter. A Component-based Specification Approach for Embedded Systems using FDTs. *ACM SIGSOFT Softw. Eng. Notes*, 31(2):14–18, 2006.
88. Giuseppe Di Guglielmo, Franco Fummi, Cristina Marconcini, and Graziano Pravadelli. FATE: a Functional ATPG to Traverse Unstabilized EFSMs. In *Proc. of IEEE ETS)*, pages 179–184, 2006.
89. Luigi Di Guglielmo, Franco Fummi, and Graziano Pravadelli. The role of mutation analysis for property qualification. *IEEE European Test Symposium*, 2009.
90. I.G. Harris. Fault models and test generation for hardware-software covalidation. *IEEE Design & Test of Computers*, 20(4):40–47, Jul. 2003.
91. Home Automation. URL: http://home-automation.org/.
92. Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault Injection Techniques and Tools. *Computer*, 30(4):75–82, 1997.
93. HT-LAB. VH2SC. `http://www.ht-lab.com/freeutils/vh2sc/vh2sc.html`.
94. IEEE. VHDL Analysis and Standardization Group. URL: http://www.eda.org/vhdl-200x/.
95. IEEE 802.11 Working Group. URL: http://www.ieee802.org/11/.
96. IEEE 802.3 Working Group. URL: http://www.ieee802.org/3/.
97. Open SystemC Initiative. IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual. *IEEE Std 1666-2005*, pages 1–423, 2006.
98. Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries. URL: http://ieeexplore.ieee.org, 1991.
99. ISO/DIS. Road vehicles – functional safety. www.iso.org/iso/catalogue_detail.htm.
100. Eric Jenn, Jean Arlat, Marcus Rimen, Joakim Ohlsson, and Johan Karlsson. Fault Injection into VHDL Models: The MEFISTO Tool. In *Proc. of IEEE FTCS*, pages 66–75, 1994.
101. R. Jindal and K. Jain. Verification of Transaction-Level SystemC Models Using RTL Testbenches. In *Proc. of ACM/IEEE MEMOCODE*, pages 199–203, 2003.
102. Ronald K. Jurgen. *Distributed Automotive Embedded Systems*. SAE International, 2007.
103. G. Kahn. The semantics of a simple language for parallel programming. *Info. Proc.*, pages 471–475, 74 (4), 1974.
104. W.-L. Kao and R. K. Iyer. DEFINE: a distributed fault injection and monitoring environment. In *Proc. of IEEE FTPDS*, pages 252–259, 1995.
105. Hisaaki Katagiri, Keiichi Yasumoto, Akira Kitajima, Teruo Higashino, and Kenichi Taniguchi. Hardware Implementation of Communication Protocols Modeled by Concurrent EFSMs with Multi-Way Synchronization. In *Proc. of ACM/IEEE DAC*, pages 762–767, 2000.
106. Ahmed Khoumsi, Gregor von Bochmann, and Rachida Dssouli. Protocol synthesis for real-time applications. In *FORTE XII / PSTV XIX '99: Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, pages 417–433, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.
107. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–277, Aug. 2000.
108. K.T.Cheng and A.S. Krishnakumar. Automatic generation of functional vectors using the extended finite state machine model. *ACM Trans. on Design Automation of Electronic Systems*, 1(1):57–79, 1996.
109. S. Kyushik. Fault simulation with the parallel valued list algorithm. In *VLSI Syst. Design*, pages 36–43, Dec. 1985.

110. Clifton Labs. AIRE/CE. http://www.cliftonlabs.com/vhdl/savant.html.
111. LAN/MAN Standards Committee of the IEEE Computer Society. Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs). *IEEE Standard for Information technology*, Sept. 2006.
112. Y. M. Levendel, P. R. Menon, and S. H. Patel. Parallel fault simulation using distributed processing. In *Bell Syst. Tech. J.*, volume 62, no. 10, pages 3107–3130, Dec. 1983.
113. F. J. Lin, P. M. Chu, and M. T. Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. In *Proc. of the ACM workshop on Frontiers in Computer Communications Technology*, pages 126–135, New York, NY, USA, 1988. ACM.
114. N. Lugil and L. Philips. A W-CDMA transceiver core and a simulation environment for 3GPP terminals. In *Proc. IEEE Symp. on Spread Spectrum Techniques and Applications*, volume 2, pages 491–495, Sept. 2000.
115. C. López, T. Riesgo, Y. Torroja, E. de la Torre, and J. Uceda. A method to perform error simulation in VHDL. *Design of Circuits and Integrated Systems Conference*, 1998.
116. S. McCanne and S. Floyd. NS Network Simulator – version 2. URL: http://www.isi.edu/nsnam/ns.
117. P. R. Moorby. Fault simulation using parallel valued lists. In *Proc. IEEE Int. Conf. Computer-Aided Design*, pages 101–102, 1983.
118. Luca Mottola, Animesh Pathak, Amol Bakshi, Viktor K. Prasanna, and Gian Pietro Picco. Enabling scope-based interactions in sensor network macroprogramming. In *Internatonal Conference on Mobile Adhoc and Sensor Systems*, pages 1–9. IEEE, October 2008.
119. N. Bombieri, F. Fummi, D. Quaglia. System/network design space exploration based on tlm for networked embedded systems. *ACM Transactions on Embedded Computer Systems*, 9(4), March 2010.
120. W. Najjar and J.-L. Gaudiot. Network resilience: A measure of network fault tolerance. *IEEE Trans. on Computers*, 39(2):174–181, Feb. 1990.
121. Nicolas Navet and Francoise Simonot-Lion, editors. *Automotive Embedded Systems Handbook*. CRC Press, 2008.
122. T. M. Niermann, W. T. Cheng, and J. H. Patel. PROOFS: A fast memory-efficient sequential circuit fault simulator. In *IEEE Trans. Computer Aided Design*, volume I-1, pages 198–207, Feb. 1992.
123. Object Management Group. Marte. URL: http://www.omgmarte.org.
124. Object Management Group. Sysml. URL: http://www.sysml.org.
125. Object Management Group. Unified modeling language. URL: http://www.uml.org.
126. R. Oboe and P. Fiorini. A design and control environment for internet-based telerobotics. *The International Journal of Robotics Research*, 17(4):433–449, 1998.
127. P. Odgaard, J. Stoustrup, P. Andersen, and E. Vidal. Accommodation of repetitive sensor faults applied to surface faults on compact discs. *IEEE Trans. on Control Systems Technology*, 16(2):348–335, March 2008.
128. Gerard O'Driscoll. *The Essential Guide to Digital Set-Top Boxes and Interactive TV*. Prentice Hall, 1999.
129. A. Jefferson Offutt. A practical system for mutation testing: help for the commonprogrammer. In *International Test Conference*, pages 824–830, Oct. 1994.
130. A. Jefferson Offutt and Ronald H. Untch. *Mutation 2000: uniting the orthogonal*. Kluwer Academic Publishers, 2001.
131. U. Y. Ogras and R. Marculescu. Energy- and performance-driven noc communication architecture synthesis using a decomposition approach. In *IEEE Conference and Exhibition on Design, Automation and Test in Europe*, 2005.
132. Open Mobile Terminal Platform. URL: http://www.omtp.org/.
133. OSTATIC. VHDLC. http://ostatic.com/vhdlc.
134. P. Levis et al. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. 1st ACM Conference on Embedded Networked Sensor Systems*, 2003.

135. DongKook Park, Chrysostomos Nicopoulos, Jongman Kim, N. Vijaykrishnan, and Chita R. Das. Exploring fault-tolerantnetwork-on-chip architectures. *International Conference on Dependable Systems and Networks*, pages 93–104, 2006.
136. P. C. Patton. Multiprocessors: Architectures and applications. *Computer*, 18(6):29–40, 1985.
137. P. Penil, F. Herrera, and E. Villar. Formal foundations for marte-systemc interoperability. In *Proc. of Forum on Design Languages (FDL)*, 2010.
138. P. Penil, J. Medina, H. Posadas, and E. Villar. Generating heterogeneous executable specifications in systemc from uml/marte models. *Innovations in Systems and Software Engineering*, 6:65–71, 2010.
139. Pablo Penil, Hector Posadas, and Eugenio Villar. Formal modeling for uml/marte concurrency resources. *Engineering of Complex Computer Systems, IEEE International Conference on*, 0:343–348, 2010.
140. G. Pfister. The Yorktown Simulation Engine. In *Proc. of 19th ACM/IEEE Design Automation Conf.*, 1982.
141. Politecnico di Torino. ITC-99 Benchmarks, 1999. `http://www.cad.polito.it/tools/itc99.html`.
142. Robert L. Probert and Kassem Saleh. Synthesis of communication protocols: Survey and assessment. *IEEE Trans. Comput.*, 40(4):468–476, 1991.
143. R. Hunt. A review of quality of service mechanisms in IP-based networks – integrated and differentiated services, multi-layer switching, MPLS and traffic engineering. *Computer Communications*, 25(1):100–108, January 2002.
144. R. Pasko et al. Functional verification of an embedded network component by co-simulation with a real network. In *Proc. IEEE HLDVT*, pages 64–67, 2000.
145. E Rijpkema, KGW Goossens, A Radulescu, and J Dielissen. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. In *IEEE Computers and Digital Techniques*, 2003.
146. Kay Rmer. Programming paradigms and middleware for sensor networks. In *URL: http://citeseer.ist.psu.edu/666689.html*, 2004.
147. Christian Sauer, Matthias Gries, and Hans-Peter Lob. Systemclick – a domain-specific framework for early exploration using functional performance models. In *Proc. of IEEE Design Automation Conference (DAC)*, pages 480–485, 2008.
148. Francesco Sechi, Luca Fanucci, Stefano Luschi, Simone Perini, and Matteo Madesani. Design of a distributed embedded system for domotic applications. In *Proc. of the 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools (DSD '08)*, pages 427–431, 2008.
149. S. Seshu. On an Improved Diagnosis Program. *IEEE Trans. on Electronic Computers*, EC-12(2):76–79, Feb. 1965.
150. M. Sida, R. Ahola, and D. Wallner. Bluetooth transceiver design and simulation with vhdl-ams. *Circuits and Devices Magazine, IEEE*, 19(2):11 – 14, March 2003.
151. C. Silvano, V. Zaccaria, G. Palermo, G. Mariani, and F. Castro. Multicube explorer. URL: http://home.dei.polimi.it/zaccaria/multicube_explorer_v1/Home.html.
152. J. Stoustrup and H. Niemann. Fault detection for nonlinear systems - a standard problem approach. In *IEEE Conf. on Decision and Control*, volume 1, pages 96–101, 1998.
153. W. Stoye, D. Greaves, N. Richards, and J. Green. Using RTL-to-C++ Translation for Large SoC Concurrent Engineering: A Case Study. In *IEEE Electronics Systems and Software*, 2003.
154. A. Sudnitson. Register Transfer Low Power Design based on Controller Decomposition. In *Proc. of IEEE MIEL*, pages 735–738, 2004.
155. A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 2003.
156. The MathWorks, Inc. Simulink. URL: http://www.mathworks.com/products/simulink/.
157. The MathWorks, Inc. Stateflow. URL: http://www.mathworks.com/products/stateflow/.

158. E. W. Thompson and S. A. Szygenda. Digital logic simulation in a time-based, table-driven environment: part 2. parallel fault simulation. In *Comput.*, volume 8, pages 38–49, Mar. 1975.
159. Transaction Level Modeling Working Group. OSCI TLM 2.0. URL: http://www.systemc.org, 2006.
160. E. G. Ulrich and T. G. Baker. Concurrent Simulation of Nearly Identical Digital Networks. *Computer*, 7(4):39–44, April 1974.
161. University of Tuebingen. VHDL-to-SystemC-Converter. `http://www-ti.informatik.uni-tuebingen.de/~systemc/`.
162. V. Aue et al. Matlab based codesign framework for wireless broadband communication DSPs. In *Proc. IEEE ICASSP*, volume 2, pages 1253–1256, 2001.
163. V. Shnayder et al. Simulating the power consumption of large-scale sensor network applications. In *Proc. SENSYS*, pages 188–200, 2004.
164. VeriPool. Verilator. `http://www.veripool.org/wiki/verilator`.
165. P. Volgyesi and A. Ledeczi. Component-based development of networked embedded applications. In *Proc. of Euromicro conference*, pages 68–73, 2002.
166. Wholesale Applications Community. URL: http://www.wacapps.net.
167. A. S. Willsky. A survey of design methods for failure detection in dynamic systems. *Automatica*, 12:601–611, 1976.
168. Hirozumi Yamaguchi, Kozo Okano, Teruo Higashino, and Kenichi Taniguchi. Protocol synthesis from time petri net based service specification. *Parallel and Distributed Systems, International Conference on*, 0:236, 1997.
169. Fred L. Yang and Resve A. Saleh. Simulation and analysis of transient faults in digital circuits. *IEEE Journal of Solid-State Circuits*, 27(3):258–264, 1992.
170. Y.X. Zhang, K. Takahashi, N. Shiratori, and S. Noguchi. An interactive protocol synthesis algorithm using a global state transition graph. *IEEE Transactions on Software Engineering*, 14:394–404, 1988.
171. ZigBee Alliance. URL: http://www.zigbee.org/.
172. Abdelkrim Zitouni, Sami Badrouchi, and Rached Tourki. Communication Architecture Synthesis for Multi-bus SoC. *Journal of Computer Science*, 2(1):63–71, 2006.