

Giovanni Perbellini

A Middleware-centric Design Methodology for Networked Embedded Systems

Ph.D. Thesis

April 14, 2009

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
prof. Franco Fummi

Series N°: **TD-05-09**

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

*a mia mamma,
la mia forza e la mia guida*

Contents

1	Introduction	1
1.1	Thesis objective	2
1.2	Structure of the thesis	4
2	Background and related works	7
2.1	NES requirements	8
2.2	Middleware structure and functionality	9
2.3	Middleware classification	11
2.3.1	Database	11
2.3.2	Tuplespace	12
2.3.3	Object-oriented (OOM)	12
2.3.4	Message-oriented (MOM)	13
2.4	NES-design state of the art	13
3	AME Design methodology	15
3.1	Typical NES design flow	15
3.2	Abstract Middleware Environment	17
3.2.1	Database Services	17
3.2.2	Tuplespace Services	17
3.2.3	Object oriented Services	18
3.2.4	Message oriented Services	19
3.3	AME implementation	19
3.4	AME-centric design flow	20
3.4.1	Refinement and Simulation	20
3.4.2	Translation	22
3.4.3	Mapping	23
4	Refinement and Simulation	25
4.1	HW/SW/Network simulation	26
4.1.1	Software	26
4.1.2	Hardware	26
4.1.3	Network	27
4.1.4	System/Network co-simulation	27

- 4.1.5 HW/SW (SystemC/ISS) co-simulation 38
- 4.1.6 HW/SW (SystemC/QEmu) co-simulation 43
- 4.2 AME_2 design level 52
 - 4.2.1 Network Simulator Interface 53
 - 4.2.2 AME-Transactor 54
 - 4.2.3 SCNSL-AME-Transactor 55
 - 4.2.4 NS2-AME-Transactor 56
- 4.3 AME_1 58
- 4.4 Actual or Simulated platform 59
- 4.5 Experimental analysis 59

- 5 Translation 69**
 - 5.1 AME Proxy Middleware 69
 - 5.2 Tuplespace to Database 70
 - 5.3 Database to Tuplespace 72
 - 5.4 Object-oriented to Tuplespace 74
 - 5.5 Tuplespace to Object-Oriented 75
 - 5.6 Tuplespace to Message-oriented 78
 - 5.7 Message-oriented to Tuplespace 79
 - 5.8 Message-oriented to Database 80
 - 5.9 Database to Message-oriented 81
 - 5.10 Message-oriented to Object-oriented 82
 - 5.11 Object-oriented to Message-oriented 83
 - 5.12 Object-oriented to Database 84
 - 5.13 Database to Object-oriented 84
 - 5.14 Experimental analysis 85

- 6 Mapping 93**
 - 6.1 Mapping onto ZigBee/Z-Stack 94
 - 6.1.1 Object-Oriented AME 94
 - 6.1.2 ZigBee 97
 - 6.1.3 Z-Stack Execution Model 98
 - 6.1.4 Methodology 99
 - 6.1.5 Experimental analysis 103
 - 6.2 Mapping onto TeenyLime MW 105
 - 6.2.1 TeenyLime application 105
 - 6.2.2 Methodology 106

- 7 Application to a heterogenous NES 109**

- 8 Conclusion 115**

- References 117**

Introduction

Ambient intelligence, pervasive and ubiquitous computing are the center of a great deal of attention because of their promise to bring benefits for end-users, higher revenues for manufacturers and new challenges for researchers. Typical computing technologies (such as telemedicine, manufacturing, crisis management) are part of a broader class of Networked Embedded Systems (NESs) in which a large number of nodes are connected together and collaborate to perform a common task under a defined set of constraints. Therefore, the key aspects of these applications are their distributed nature and the presence of very limited HW resources, as in case of Wireless Sensor Networks (WSNs).

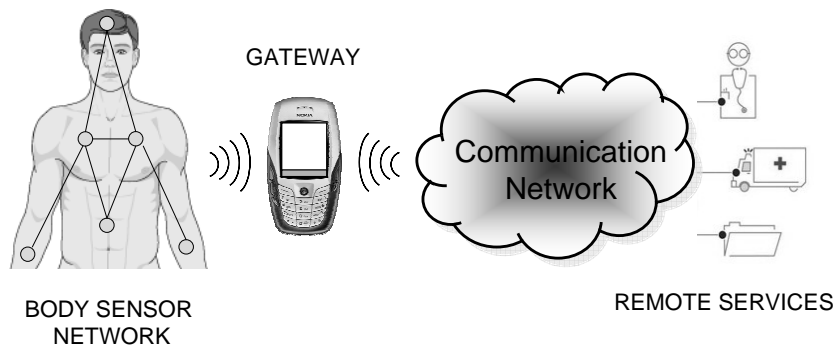


Fig. 1.1. Health-care application of NES.

Figure 1.1 illustrates an example of health-care application in which a body sensor network monitors patient's parameters (e.g., temperature, blood pressure, and motion) and transmits them through the Gateway to a Remote Service to control/monitor the user health (e.g., hospital).

In the development of NES software, re-use of components to speed-up the time-to-market contrasts with the need of ad-hoc solutions for the limited HW

resources. Simulation plays a key role in this development process to allow not only functional validation but also design-space exploration.

A reuse-driven approach to embedded software for NES must emphasise several key characteristics [14]:

- use of standard programming paradigms to write applications and standard communication paradigms between nodes to guarantee interoperability across different manufacturers;
- presence of a well-defined abstraction layer which isolates application from the details of the underlying platform to simplify the NES application development;
- simulation tools for functional validation and the fulfilment of tight HW/SW constraints and efficient modelling techniques to explore different design solutions.

Traditionally, many NES applications have been developed without support from system software [15]. When system software has been used it has consisted of simple device drivers and an operating system. State-of-the-art techniques [16] for NES focus on simple data-gathering applications, and in most cases, the design of the application and the system software are usually closely-coupled, or even combined as a monolithic procedure. However, such procedures are sometimes ad-hoc and impose direct interaction with the underlying embedded operating system, or even the hardware components. Such applications are neither flexible nor scalable and they should be re-written if the platform changes.

Due to these problems, Networked Embedded Systems make application development non-trivial. Middleware is emerging as an important architectural component in supporting NES applications able to facilitate the application development. The role of middleware is to present a unified programming model to application designers and to mask out the problems of heterogeneity and distribution providing a basic set of tools and libraries for the low-level handling of technology-specific NES. It represents a service layer which abstracts from the peculiarities of the operating system and HW components. Several NES middlewares have been implemented in the past [17,18], each one providing different programming paradigms (e.g., tuplespace, message-oriented, object-oriented, database, etc.). Nowadays, the choice of the middleware to design NES application is based on the following criteria: programming skills of the system architect [19] and platform constraints [20].

The down-side of this traditional approach is a more complex NES application design flow. In fact, the same application cannot support different platforms, limiting application re-use, interoperability and scalability.

1.1 Thesis objective

Despite of these punctual contributions, the literature does not report a complete design methodology for NES applications integrating interoperability, simulation and simplification aspects.

The goal of this thesis is to present a middleware-centric design methodology for NES, where the middleware plays a decisive role in the design process. The proposed methodology allows:

- To build the application over a middleware-like service layer named *abstract middleware* hiding the different NES implementations peculiarities from end-user applications; this feature is reached by the Abstract Middleware Environment (AME), a framework that abstracts common programming paradigms to design NES applications: tuplespace, publish-subscribe (MOM), object-oriented (OOM) and database.
- To choose for the application development the programming paradigm that maximises productivity and to explore different design solutions by automatically translating application code to use another programming paradigm in order to satisfy system's functional and non-functional properties (e.g., synchronous or asynchronous communication, tight or loosely coupled interaction, etc.). For instance, an asynchronous communication requirement would not be satisfied by using Tuplespace. Similarly, using a MOM for developing a messaging application implies to explicitly deploy a message broker component and to interact with it for sending and receiving messages.
- To simulate the application for functional validation supporting interoperability between different implementation platforms and ensure scalability of the NES technology.

AME provides a complete framework to design and simulate networked embedded systems application. This environment allows to design applications through three design step:

- The first step (named AME_3), depicted in Figure 1.2.a, simulates and validates application functional requirements by using middleware-like services with different programming paradigms.
- During the second step (AME_2), a simulated network infrastructure is involved in the whole framework, as shown in Figure 1.2.b. AME_2 provides the same APIs of the previous step, even if opportunely modified to establish a communication with a network simulator (e.g., NS2 [53], SCNSL [73]).
- At the third step (called AME_1) HW/SW partitioning is applied to each node to map functionalities to HW and SW components accordingly to several constraints (e.g., performance, cost, and component availability). At this level communication APIs provided by AME_1 services are the same APIs of the previous step (AME_2). HW components are involved in the simulation through HW/SW/network simulation (Figure 1.2.c); in this case the application code can be changed for the modelling of the HW components.

The described design flow provides three main advantages:

- application development is simplified since the choice of the programming paradigm is not constrained by deployment issues;
- application development and platform design can be performed concurrently;
- design-space exploration is improved since it is possible to evaluate the performance of the same application implemented by using different programming paradigms.

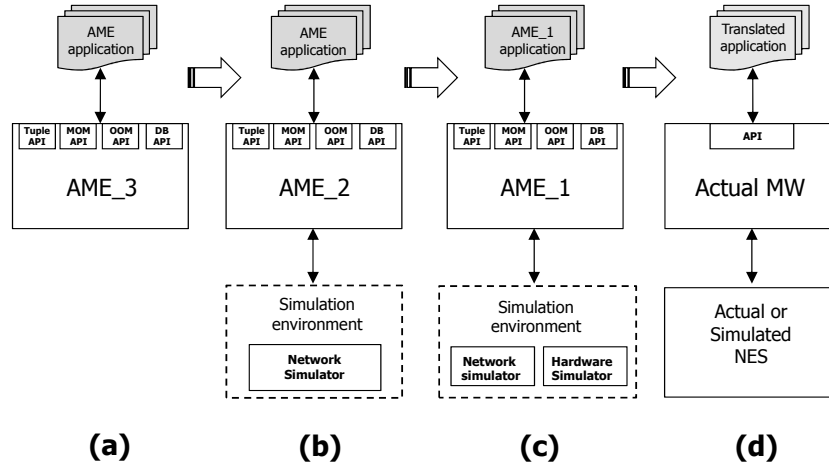


Fig. 1.2. AME design flow.

1.2 Structure of the thesis

This thesis is organized as follows.

In Chapter 2 we present the different proposals which have been done to solve the NES application issues previously described. In this context we present some works providing abstraction with respect to the platform model or the architectural styles (e.g., programming paradigms). Moreover, in this Chapter we classify the NES middleware approaches according to their programming paradigms.

Chapter 3 describes the middleware-centric design methodology highlighting the presence of the middleware introduced as an explicit design dimension with respect to the typical NES design flow. Furthermore, the AME environment and the related API set involved for NES application design is described. The AME-centric design flow has been published on [2] and [3].

Chapter 4 reports the AME *Refinement* process concealing the peculiarities of the underlying NES, where the simulation environment is involved in order to simulate the NES applications taking in account network and hardware effects. Moreover, the whole simulation environment integrated in AME is described. The simulation environment is composed by two main co-simulation environments: System/Network and HW/SW co-simulation. Finally, the *Refinement* process has been applied to a NES-based application in order to describe the advantages of the AME design methodology. The co-simulation environment has been published

on [1], [4], [5], [7], [8], [10] and [13]. Moreover the *Refinement* process has been published on [12].

Chapter 5 focuses on the translation between the different AME programming paradigms and reports the methodology to translate application code. Finally, the proposed translation mechanism has been applied to a NES application scenario to evaluate the effectiveness of the proposed solution. The reference application has been designed by using the AME programming paradigms: TupleSpace, Object-Oriented, Database, Message-oriented. The pseudo-code for each implementation is described. The translation mechanism has been published on [9].

Chapter 6 describes the mapping process from the AME application to the actual NES by mapping AME calls to actual middleware calls. Two reference examples are presented: the description of the automatic mapping of AME applications over a target NES platform running Z-Stack middleware (ZigBee) and the *Mapping* process onto a TupleSpace programming paradigm (named TeenyLime). Finally, experimental results are reported to show the advantages of the AME-centric design methodology. The automatic mapping process has been published on [11].

Chapter 7 shows the modelling of a real Networked Embedded Systems, named Angel platform. This chapter emphasizes how the AME provides an environment supporting the development of applications, as well as the analysis and optimisation of the interactions among the platform components. The Angel platform has been partially described in [6].

Background and related works

A Middleware layer is a novel approach to fully meeting the design and implementation challenges of NES applications. Middleware has often been useful in traditional systems for bridging the gap between the operating system (a low-level component) and the application (as shown in Figure 2.1), easing the development of the distributed applications.

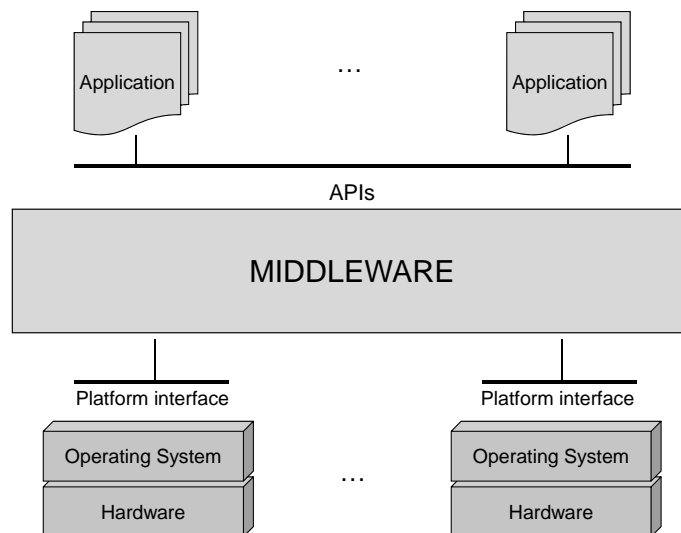


Fig. 2.1. Middleware layer.

All services provided by a middleware system should respect the constraints involved in NES, which are limited amount of memory, reduced processing power, scalability, heterogeneity. Several middleware systems have been designed to deal with the previous issues. Next Section presents an exhaustive analysis and classi-

fication of the middleware for NES. Finally, we propose to classify them in four main classes.

2.1 NES requirements

We envision that the development of NES will finally demand systematic application design methods based on standard and portable abstractions of the system. Thus, middleware sitting between the hardware, operating systems and the application is required to provide:

- standardized system services to different applications.
- a runtime environment that can support and coordinate multiple applications.
- mechanisms to achieve adaptive and efficient utilization of system resources.

Such a middleware is particularly useful for NES that host complex applications with large amount of information processing and/or stringent performance constraints. The NES application design and development through a middleware-based approach, must address many requirements dictated by NES characteristics. Following we describe the requirements which should be satisfied by a middleware-centric design flow.

Heterogeneity

NES applications (such as industrial machines, medical equipment, household appliances, mobile phones, PDAs, sensors and actuators) written in different programming languages (C, C++, nescC and Java) running on different operating systems, executing on different hardware platforms, should be able to communicate using a middleware platform. As possible, middleware system should include the necessary abstractions in order to cater for the heterogeneous nature of a network embedded environment consisting of different types of devices, but cooperating with the middleware. Moreover, the middleware system should include the flexibility to use the available communication protocols that are eventually supported by particular devices.

Power and resources

Limited in energy and individual resources (such as CPU and memory), these tiny devices could be deployed in hundreds or even thousands in harsh and hostile environments. In cases where physical contact for replacement or maintenance is impossible, wireless media is the only way for remote accessibility. Hence, middleware should provide mechanisms for efficient processor and memory use while enabling lower-power communication. A NES should accomplish its three basic operations sensing, data processing, and communication without exhausting resources.

Openness

Implementation of new functionality, or changes of an existing functionality should be possible to be permitted within the middleware as the set of applications changes or the set of embedded nodes is updated with new nodes, offering new functionality to the application. Therefore, as in the case of any distributed system, the middleware should have the capability to be extended and modified during its lifetime. Moreover, since data should be continuously be provided to the

application, especially in the case of real-time applications, the process of updating or extending the middleware should not require halting its operation while this is being done.

Scalability and mobility

If an application grows, the network should be flexible enough to allow this growth anywhere and anytime without affecting network performance, so this is the scalability. Moreover efficient middleware services must be capable of maintaining acceptable performance levels as the network grows. Network topology is subject to frequent changes owing to factors such as malfunctioning, device failure, moving obstacles, mobility, and interference. Middleware should support NES robust operation despite these dynamics by adapting to the changing network environment. Middleware also should support mechanisms for fault tolerance and NES self-configuration and self-maintenance.

Development time

A middleware should provide the mechanisms to reduce the amount of time and effort required to build a system application.

Data aggregation

Often in a NES the applications involves nodes which both provide redundant data and are locate in specific local region. Then data aggregation open the possibility to in-network aggregation from different sources erasing the redundancy and reducing the number of transmission. This technique allows save energy and resources.

Security

NES are being widely deployed in domains that involve sensitive information; typically, NES uses wireless medium facilitating unwanted packets injection to compromise the network's functioning. All these factors make security extremely important. Furthermore, NES have limited power and processing resources, so standard security mechanisms, which are heavy in weight and resource consumption, are unsuitable. These challenges increase the need to develop comprehensive and secure solutions that achieve wider protection, while maintaining desirable network performance.

2.2 Middleware structure and functionality

Networking protocol stacks can be decomposed into multiple layers such as the physical, data-link, network, transport, session, presentation, and application layers. Similarly, middleware can be decomposed into multiple layers such as those shown in Figure 2.2.

We describe each of these middleware layers and outline some of the commercial-off-the-shelf (COTS) technologies in each layer [21].

Host infrastructure middleware encapsulates OS concurrency and inter-process communication (IPC) mechanisms to create object-oriented network programming capabilities. These capabilities eliminate many tedious, error-prone, and non-portable activities associated with developing networked applications via native OS APIs, such as Sockets or POSIX threads (Pthreads). Examples of COTS

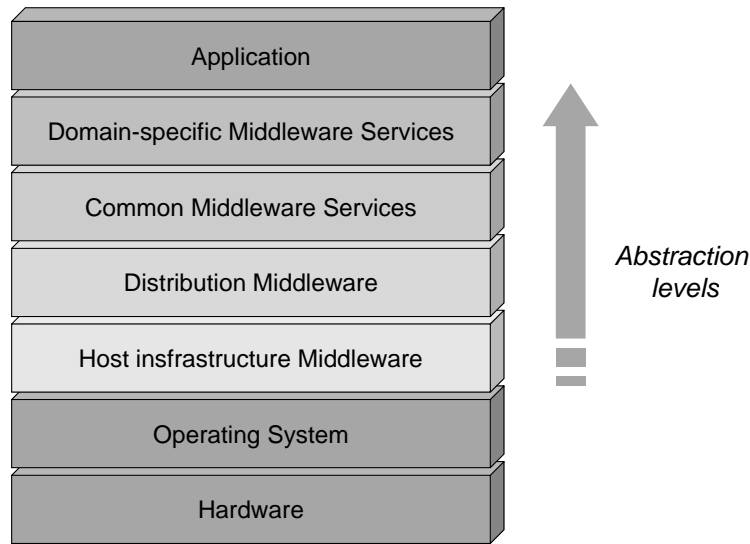


Fig. 2.2. A level-based middleware classification.

host infrastructure middleware include the following: the Adaptive Communication Environment (ACE) [22] is a portable and efficient toolkit that encapsulates native operating system network programming capabilities such as inter-process communication, static and dynamic configuration of application components, and synchronization. Real-Time Java Virtual Machines implement the Real-Time Specification for Java (RTSJ) [23]. The RTSJ is a set of extensions to Java that provide a largely platform-independent way of executing code by encapsulating the differences between real-time operating systems and CPU architectures..

Distribution middleware uses and extends host infrastructure middleware in order to automate common network programming tasks, such as connection and memory management, marshaling and demarshaling, endpoint and request demultiplexing, synchronization, and multithreading. Developers who use distribution middleware can program distributed applications much like stand-alone applications, that is, by invoking operations on target objects without concern for their location, language, OS, or hardware. At the heart of distribution middleware are Object Request Brokers (ORBs), such as Java RMI [24], and CORBA [25].

Common middleware services augment distribution middleware by defining higher-level domain-independent services, such as event notification, logging, persistence, security, and recoverable transactions. Whereas distribution middleware focuses largely on managing end-system resources in support of an object-oriented distributed programming model, common middleware services focus on allocating, scheduling, and coordinating various resources throughout a distributed system. Without common middleware services, these end-to-end capabilities would have to be implemented ad hoc by each networked application. Examples of common middleware services include the OMG's CORBAServices [26] and the CORBA

Component Model (CCM) [27], which provide domain-independent interfaces and distribution capabilities that can be used by many distributed applications. The OMG CORBAServices and CCM specifications define a wide variety of these services, including event notification, naming, security, and fault tolerance.

Domain-specific middleware services satisfy specific requirements of particular domains, such as telecommunications, e-commerce, health care, process automation, or avionics. Whereas the other object-oriented middleware layers provide broadly reusable "horizontal" mechanisms and services, domain-specific services target vertical markets. From a "commercial off-the-shelf" (COTS) perspective, domain-specific services are the least mature of the middleware layers today. This is due in part to the historical lack of middleware standards needed to provide a stable base upon which to create domain-specific services.

2.3 Middleware classification

The layer decomposition of the whole middleware architecture is a typical model-driven approach to simplify the design tasks, and classify the functionalities provided by the middleware. Moreover, almost all actual middleware are cross layer, providing a set of services at each layer. Some research efforts have been directed to the development of new middleware based on different programming paradigms [28]. A programming paradigm allows to program the NES as a standalone application - it should hide hardware and distribution issues from the programmer as far as possible. We propose to classify them in following four main classes as reported in Figure 2.3

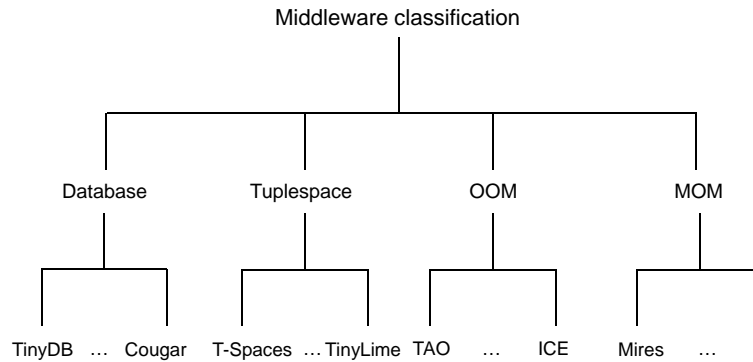


Fig. 2.3. A programming paradigm-based middleware classification.

2.3.1 Database

A number of approaches have been proposed that treat the NES as a distributed database where users can issue SQL-like queries to extract the data of interest from

the nodes of networks. Cougar [29] introduces a new dimension in middleware research by adopting a database approach in which NES data are considered a virtual relational database. Cougar implements NES management operations in the form of queries, using an SQL-like language. TinyDB [30] is a query-processing system for extracting information from a network of sensor devices using TinyOS as an operating system. TinyDB relieves the user from the complexity to write embedded code by providing an easy, SQL-like interface for extracting the data of interest from sensor nodes with limited power and hardware resources. The queries use simple data manipulation to indicate the type of readings, such as light and temperature, as well as the subset nodes of interest.

2.3.2 TupleSpace

The characteristics of wireless communication media (e.g., low and variable bandwidth, frequent disconnections, etc.) favor a decoupled and opportunistic style of communication: decoupled in the sense that computation proceeds even in presence of disconnections, and opportunistic as it exploits connectivity whenever it becomes available. The synchronous communication paradigm supported by many traditional distributed systems has to be replaced by a new asynchronous communication style. This communication problem has been addressed by TupleSpace based systems. Although not initially designed for this purpose (their origins go back to Linda [31], a coordination language for concurrent programming), TupleSpace systems have been shown to provide many useful facilities for communication in wireless settings. In Linda, a TupleSpace is a globally shared, associatively addressed memory space used by processes to communicate. It acts as a repository of data structures called tuples that can be seen as vectors of typed values. Tuples constitute the basic elements of a TupleSpace systems; they are created by a process and placed in the tuple space using a write primitive, and they can be accessed concurrently by several processes using read and take primitives, both of which are blocking (even if nonblocking versions can be provided). Tuples are anonymous, thus their selection takes place through pattern matching on the tuple contents. Communications is decoupled in both time and space: senders and receivers do not need to be available at the same time, because tuples have their own life span, independent of the process that generated them, and mutual knowledge of their location is not necessary for data exchange, as the tuple space looks like a globally shared data space, regardless of machine or platform boundaries. Some examples belonging of this class are: T-Spaces [32] and TinyLime [33].

2.3.3 Object-oriented (OOM)

Perhaps the most popular model is object based middleware in which applications are structured into (potentially distributed) objects that interact via location transparent method invocation. Object-oriented Middleware offers synchronous, typed communication between components of a distributed program. An object model is a set of definitions about the properties of computational entities, such as the available types and their semantics, rules for type compatibility, behavior in case of errors, and so on. Typically, these Middlewares offer an interface definition

language (IDL) which is used to abstract over the fact that objects can be implemented in any suitable programming language, an object request broker which is responsible for transparently directing method invocations to the appropriate target object, and a set of services (e.g. naming, time, transactions, replication etc.) which further enhance the distributed programming environment. Ice [35] is a new object-oriented middleware platform that allows developers to build distributed client-server applications with minimal effort. The ACE ORB (TAO) is an open-source standard-compliant implementation of CORBA that's optimized for high-performance and real-time systems; it allows clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware [36].

2.3.4 Message-oriented (MOM)

Message-oriented middleware supports asynchronous calls between the client and server applications. MOM increases the flexibility of an architecture by enabling applications to exchange messages (containing formatted data, requests for action, or both) with other programs without having to know what platform or processor the other application resides on within the network. Nominally, MOM systems provide a message queue between interoperating processes, so if the destination process is busy, the message is held in a temporary storage location until it can be processed. MOM is typically asynchronous and peer-to-peer, but most implementations support synchronous message passing as well. This approach is quite suitable in pervasive environments such as NES, where most applications are based on events. It adopts a component-based programming model using active messages to implement its publish-subscribe-based communication infrastructure. In this programming model, sources "publish" to the entire network and interested sinks "subscribe" to messages. The network then only forwards them downstream if there is at least one subscriber on that path. This requires the message transport service to understand the message internals, although some systems are "topic-based" where each message has a subject line which the transport system reads and can ignore the rest of the message. Mires [18] proposes an adaptation of a message-oriented middleware for traditional fixed distributed systems. Mires provides an asynchronous communication model that is suitable for NES applications (which are event driven in most cases).

2.4 NES-design state of the art

The diversity of properties provided by each middleware makes complex the development of high quality middleware-based software systems: software engineering methods and tools should be developed with the use of middleware in mind. The use of middleware affects the following software development phases [37]:

- The selection of the middleware to be used should be based on engineering methods and on the system requirements.
- System design, specification and analysis must integrate properties manage at middleware level.

- System implementation should be built as much as possible on middleware tools.
- System validation needs to be performed integrating both middleware-related and application specific components.

Some works try to overcome this problem by using abstraction with respect to the platform model or the architectural styles (e.g., programming paradigms).

In such way, Sensation [41] presents a middleware platform solution for pervasive applications in WSN providing a developer-friendly programming interface. This approach is valid just for WSN and does not include a network simulator for an exhaustive network evaluation.

Model-Driven Engineering (MDE) is a significant step towards a middleware-based software process [38]. The best known MDE initiative is the Model-Driven Architecture (MDA) from the Object Management Group (OMG). Using the MDA [39, 40] methodology, system functionalities may first be defined as a platform independent model (PIM) through an appropriate Domain Specific Language. Given a Platform Definition Model (PDM) corresponding to some middleware, the PIM may then be translated to one or more platform-specific models (PSMs) for the actual implementation. Translations between the PIM and PSMs are normally performed using automated tools, like Model transformation tools. The MDA focuses primarily on the functionality and behaviour of a distributed application or system, not on the technology in which it will be implemented. Furthermore, MDA does not directly provide a simulation environment.

PrismMW [42] is an extensible middleware platform that enables implementation, deployment and execution of distributed Prism (Programming in the small and many) applications in terms of their architectural elements: components, connectors, configurations, and events. The key properties of Prism-MW are its native, and flexible, support for architectural abstractions (including architectural styles), efficiency, scalability, and extensibility. PrismMW allows to develop Java or C++ applications running on Java Virtual Machine or Windows CE respectively.

A Universal Middleware Bridge (UMB) system has been proposed in [43] to solve the interoperability problem caused by the heterogeneity of several types of home network middleware. UMB system makes middleware interoperable with another middleware by using a device conversion mechanism and a message translation mechanism. This approach introduces a new huge software layer (the UMB) between the actual middleware and the application; therefore this solution is not feasible in networked embedded systems where the HW/SW resources available are very poor.

[44] supports true interoperability between different applications, as well as between different implementation platforms and ensure scalability of the NES technology by proposing a universal application interface, which allows programmers to develop applications without having to know unnecessary details of the underlying platform. Such works define a standard set of services and interface primitives called SNSP (Sensor Network Services Platform), to facilitate the development of Wireless Sensors/Actuators Network applications. Unfortunately, this abstraction model of middleware sensibly simplifies the complexity of developing applications only for WSNs, merging an abstract implementation of the main services provided by actual WSN middleware.

AME Design methodology

A typical NES application is a distributed application composed by a set of interactive modules running on a heterogeneous HW/SW embedded platform (network nodes interact through communication network) to carry out the functionality. A milestone in the effort of simplifying the implementation of such applications has been the introduction of a service layer, named middleware, which abstracts from the peculiarities of the operating system and HW components. However, the presence of the middleware has not been yet introduced in the design flow as an explicit dimension. This Section introduces before the typical NES design flow and following describes the design methodology based on an Abstract Middleware Environment (AME) that allows to abstract common programming paradigms.

3.1 Typical NES design flow

Figure 3.1 shows the typical NES design flow; it starts from the specification of application requirements both functional and non-functional (e.g., cost, speed, area, and power consumption). From these requirements, a model of the whole distributed application is build as the interaction of modules connected by communication channels. Communication primitives are provided by the modelling tool and, at this stage, there is no distinction between intra-node communications and inter-node communications. Among various system modelling languages, SystemC [45] can be used for its great flexibility in describing systems at different abstraction levels and for its support of Transaction Level Modeling (TLM) [46] whose communication primitives can be employed at this stage of the design flow.

Then System/Network partitioning is applied to this model to map modules onto network nodes; an integer number of modules can be assigned to each node. We refer to the model of each node as system model. Communications between modules belonging to different nodes are now described as network communications and are part of the network model which reproduces the behaviour of network protocols (e.g., TCP/IP, ZigBee/802.15.4, etc.). System and network models can be simulated by different tools even if they should interact (horizontal arrow in the Figure 3.1) to exchange data and to share a common simulation time scale.

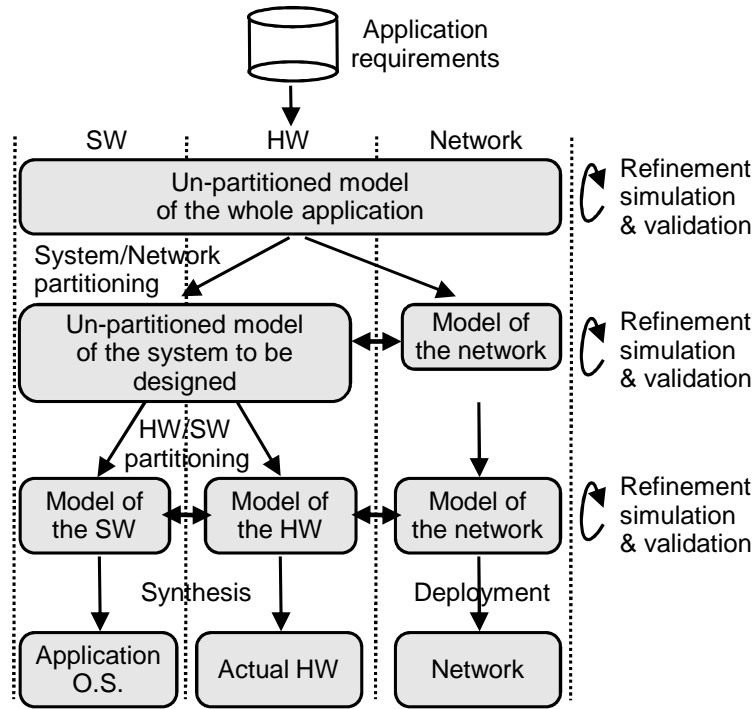


Fig. 3.1. Typical NES design flow.

In the next stage, system models which are outside the design scope remain unchanged while a traditional design flow can be applied to each system model to be developed (e.g., transaction level modelling). In particular, HW/SW partitioning is performed on the system model to map functionalities onto HW and SW components according to several constraints (e.g., speed, cost, and component availability). HW components outside the design scope remain unchanged while others are refined down to RTL and gate level. In this phase simulation can be done with different tools for SW,

HW and network components but they should cooperate to exchange data and to share a common simulation time scale. In particular, SW components interact with HW components and HW components interact with the network model (horizontal arrows in the Figure 3.1); the simulation of interactions between HW and SW components is exactly the scope of simulation environment that will be described in Section 4), while for interactions between HW and network models the same techniques of the previous design phase.

In the final stage HW and SW models are mapped onto actual components. HW components can be either synthesized or taken from the market. For what concerns software, an actual operating system is introduced and SW functionalities become application code with calls to the OS services. An actual network is used according

to its model. It is worth remarking that refinement, simulation and validation can be accomplished at each step of the design flow.

A NES application is a distributed application running on heterogeneous HW/SW embedded architecture interacting through communication network. NES systems are distributed, but the nodes must achieve a centralized goal cooperatively. Following the design flow proposed in this Section, it is possible to design NES applications without any support from system software. Moreover, NES nodes typically have limited computing power and small amounts of memory. They must consume as little power as possible. Communication is noisy and its bandwidth is limited. The individual nodes and communication channels are inherently unreliable, yet the overall system needs to be robust. These requirements, already described in Section 2.1, mandate novel systems and software design techniques.

A milestone in the effort of simplifying the implementation of such applications is the introduction of a service layer, named middleware, which abstracts from the peculiarities of the operating system and HW components. This work presents a middleware-centric design flow, where the middleware is an explicit design dimension.

3.2 Abstract Middleware Environment

Any middleware type provides different services (or APIs) to the application designer. Apart from core features, almost all middlewares, presented in Section 2.3, provide several services that simplify the application development by providing access to particular operating systems and hardware functionality. Let us list and organize the main services provided by the middleware.

3.2.1 Database Services

Database middleware approach provides the user with a query system which is very easy to use. The database approach hides distribution issues from the user. We can summarize the services provided by the Database middleware in a unique service:

`query` : SQL-like queries to perform Networked Embedded Systems tasks.

The following pseudo-code represents a simplified version of an application based on database middleware.

```
string request="SELECT temperature
                FROM table
                WHERE temperature>40"
response = middleware.query(request);
```

3.2.2 Tuplespace Services

In this middleware the data are represented by structures called tuples. They are collected in a globally shared memory called tuple space. Each tuple is a sequence

of values that also can be of different type (e.g. `i` "temperature", 25, "XYZ"`i`) and the communication between process is implemented with reading, writing and extraction of this elements from the space of tuples. The main services provided by the programming paradigm tuples-based are:

`read_b(template)` : a blocking read operation to read a tuple from *tuple space* without extract the tuple and suspending the operation until a matching tuple appears.

`read_nb(template)` : like the previous service, but in this case the operation return `null` if no matching *tuples* exists in the *tuple space*.

`take_b(template)` : a blocking operation to remove a tuple from *tuple space* suspending the operation until a matching tuple appears.

`take_nb(template)` : like the previous service, but in this case the operation return `null` if no matching *tuples* exists in the *tuple space*.

`write(tuple)` : to perform an adding *tuple* operation on the *tuple space*.

All operations, except `write` service, specify a `template` to read or extract a *tuple* from the *tuple space*. The template itself is a tuple whose field contain either values (actuals) or "wild cards" (formals). Typically, if multiple tuples match a template, the one returned by the read or take operations is selected non-deterministically.

The following pseudo-code represents a simplified version of an application based on tuplespace middleware.

```
while(1) {
    response = middleware.take_nb
        (<"temperature", ?>);
    if (response!=NULL AND
        response[1] > 40)
        execute_operation;
    endif;
}
```

3.2.3 Object oriented Services

A typical object-oriented middleware provides 1) a mechanism to describe an object interface and to map it to an actual object implemented in common programming languages, 2) a mechanism to provide the client with a local reference of the remote object, and 3) a public repository in which instances of the actual object have been registered. Let `Srv` and `SrvImpl` be the name of the object interface and of its actual implementation respectively, then the middleware should provide the following services:

`register(obj,name)` : to register an instance of `SrvImpl` into the public repository and to assign it a public name.

`lookup(name)` : to obtain a local object reference of type `Srv`.

The following pseudo-code represents a fragment of application which uses the service provided by a remote object.

```

Middleware mw=new Middleware(host,port);
Srv mySrv=(Srv)mw.lookup("Service");
mySrv.do();

```

The following pseudo-code represents a fragment of an application which creates an instance of `SrvImpl` and registers it in the public repository.

```

Middleware mw=new Middleware(host,port);
SrvImpl mySrvImpl=new SrvImpl();
mw.register(mySrvImpl, "Service");

```

3.2.4 Message oriented Services

Publish/Subscribe is an asynchronous messaging paradigm. In a Publish/Subscribe system, publishers post messages to an intermediary broker and subscribers register subscriptions with that broker. In a topic-based system, messages are published to "topics" or named logical channels which are hosted by a broker. Subscribers in a topic-based system will receive all messages published to the topics to which they subscribe and all subscribers to a topic will receive the same messages. The current programming model uses the following services:

- `void publish(Message,Topic)`: to publish a message `Message` into the topic `Topic`;
- `void subscribe(Topic, event)`: to subscribe to a particular `Topic`;
- `void unsubscribe(Topic, event)`: to unsubscribe to a particular `Topic`.

Concerning the subscribe and unsubscribe services, the first parameter represents a topic. The second parameter is an event used by AME to wake-up the application subscriber when a message is published in that topic. An event consists of a name and a payload. The event's payload carries data information: msg and topic. Figure 3.2 depicts the communication flow between two applications (Node A and Node B) implemented by using the AME MOM programming paradigm.

3.3 AME implementation

Based on the previous analysis has been implemented a set of services and interface primitives (APIs) involved in a simulation environment called Abstract Middleware Environment (AME). These APIs should be made available to an application programmer independently on the implementation on any actual middleware. Each programming service, previously described, should be seen as a component of AME.

AME environment has been implemented using SystemC. The well-known system description language named SystemC [45] is used to write the application as a set of concurrent systems interacting together. The SystemC framework provides primitives to model concurrent processes (threads), to synchronize them and to exchange messages. Furthermore, the SystemC simulation engine can be exploited for the functional validation of the application. This allows to simulate each component of the whole NES application at different level of abstraction by using the

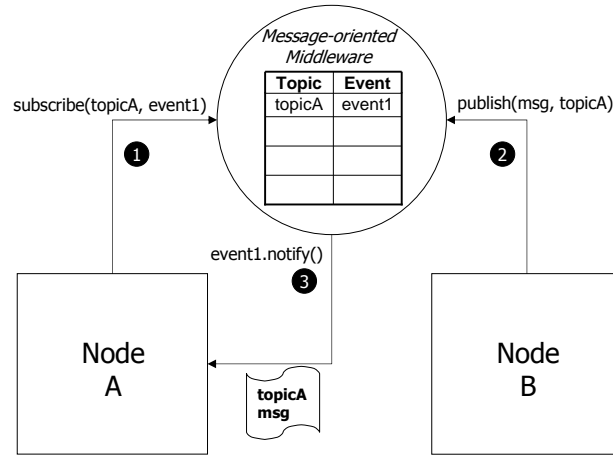


Fig. 3.2. MOM general schema.

Transaction Level Modeling (TLM) library provided by SystemC, thus providing an early platform for software development. Following the TLM fashion we have defined three AME level (AME_1, AME_2 and AME_3), usable in different abstraction levels of the application design flow, as described in Section 3.4. Each AME level includes the same API; in this way we guarantee the reuse of modeled applications at different abstraction levels

3.4 AME-centric design flow

AME offers an application interface providing the possible services, described in Section 3.2, that can be used to design a typical NES application. The AME-centric design flow consists of three key concepts, i.e., *Refinement*, *Translation*, and *Mapping*.

3.4.1 Refinement and Simulation

As depicted in Figure 3.3, the *Refinement* process can be represented by a vertical dimension in which the model of the NES platform below the application is detailed. The design flow proposed in this work is similar to the typical NES flow presented in Section 3.1. Figure 3.4 shows the design flow for NES application based on the AME, highlighting the presence of the middleware introduced as an explicit design dimension. We define three levels of detail.

At the highest level (called AME_3) modules communicate through abstract point-to-point primitives. During this step of the design flow, the designer has to specify the application requirements (performances, functionalities, power consuming, etc.) and choose the programming paradigm; a programming model should substantially support the development of the application hiding hardware and communication issues from the programmer as far as possible. Ideally a programming paradigm allows to program the networked platform as a single "virtual"

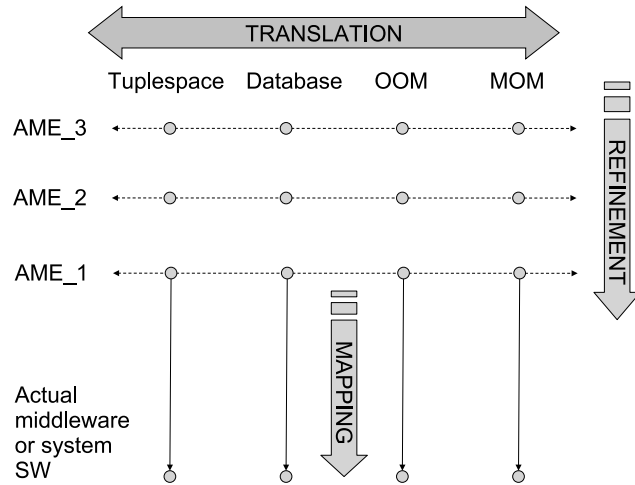


Fig. 3.3. AME-centric design flow.

entity, rather than focusing on individual nodes. The choice of the programming paradigm depends on the application designer skills and on the type/nature of the application. Therefore, in this phase the NES application will be built using the abstracted services provided by the AME_3, based on the programming paradigm chosen, to verify and simulate the functional property of the application. Furthermore, because of the separation of the application model from the middleware, the application development can be done in parallel with the design of the HW/SW platform or even without knowledge about the final platform.

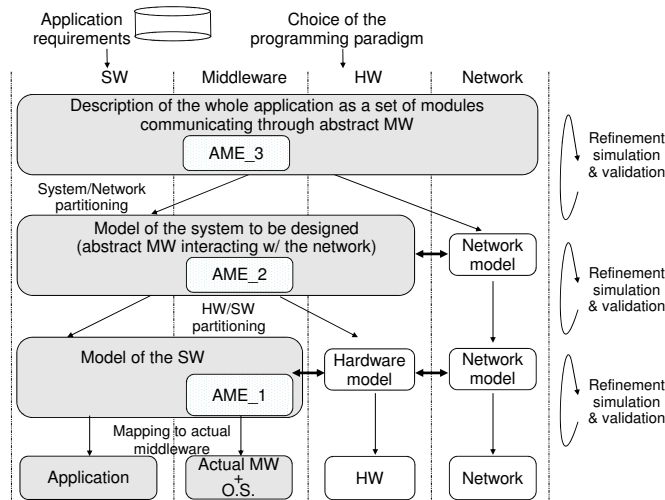


Fig. 3.4. Refinement process.

At the next level (called AME_2) *System/Network partitioning* is applied to the model. We refer to the model of the node under design as *System model*. Communications between nodes are described as network communications in the *Network model* which reproduces the behavior of network protocols such as TCP/IP or ZigBee/802.15.4. The *System model*, modeled using the services of AME_2, interacts with the *Network model* (horizontal arrow in Figure 3.4). In this phase, an performed of the communication protocols, simulated by the *Network model*, can be evaluated. The AME_2 API services used to design the NES application as the same of the previous design step; however, in this case, the implementation of the AME_2 services is different to allow the interaction with the network simulator. At the next level (called AME_1) a traditional design flow is applied to the *System model*, while the different parameters of the *Network model* can be tuned to improve performance. In particular, *HW/SW partitioning* is performed on the *System model* to map functionalities to HW and SW components according to several constraints (e.g., performance, cost, and component availability). SW components interact with HW components (named *Hardware model*) and HW components interact with the *Network model* (horizontal arrows in the Figure 3.4). Also in this case, the AME_1 API services as the same of the previous design steps, but the implementation is different to communicate with the network simulator, simulating the *Network model*.

It is worth noting that the *Refinement* process involves the model of the NES while the application software remains un-changed during the *System/Network partitioning* design process. Therefore, different development teams can work on application and NES at the same time since the interface between application and AME is well-defined.

3.4.2 Translation

The second concept is called *Translation* and it plays a key role in the AME-based design flow. It consists in modifying the application software to use a different programming paradigm without changing its functional behaviour. If programming paradigms are well defined in terms of primitives then a set of rules can be derived to transform application software from a paradigm to another. The main advantage of this translation is that programming paradigm can be changed between application development and deployment without an extra effort to re-write the application software. Usually during application development the choice of the programming paradigm is driven by the expertise of the designer and by the need to re-use application components from other projects. At deployment time, it may happen that there is no actual middleware featuring the chosen programming paradigm or this programming paradigm cannot be implemented efficiently on the actual embedded platform. The translation process gives the freedom to choose a different programming paradigm during development and deployment. Translation is performed on the abstract middleware since abstract primitives are well defined and translation rules can be derived more easily. Translation can be performed in each of the three levels of the refinement process since it involves application software not the underlying model of the embedded system. The use of *Translation* during application development can also improve design-space exploration

since the designer can also evaluate performance as a function of the programming paradigm; in fact, for a given functional requirement some paradigms can be more efficient than others in terms of resource usage (e.g., CPU, memory, network).

3.4.3 Mapping

The third concept is called *Mapping* and it regards the deployment of the application over the actual NES. If the HW and SW resources of the actual platform allow the presence of an actual middleware, then the application software is modified to replace calls to AME services with calls to the actual middleware. The *Mapping* process assumes that the programming paradigm remains the same in the transfer from abstract middleware to actual middleware; for example, an application designed as a collection of objects is mapped onto an object-oriented middleware such as CORBA. In a more general scenario, different parts of the distributed system may require different types of actual middleware and the unique abstract middleware must be mapped on those. A special case is given when the resources of the node do not tolerate the overhead of a middleware; an example is represented by the wireless sensor networks in which nodes usually have limited memory and processing power. In this case, calls to AME have to be replaced with direct calls to system SW (e.g., operating system and network stack).

Refinement and Simulation

One of key advantages of the AME-based design is that simulation can be done at the early stage of the design flow. With reference to Figure 3.4 there are different simulation mechanisms and capabilities at the different levels of the design flow (AME_3, AME_2 and AME_1). This design flow is named *Refinement* process. It conceals the peculiarities of the underlying NES, and the simulation environment is involved in order to simulate the NES applications taking in account network (AME_2 design level) and hardware (AME_1 design level) effects.

At the first stage, the whole application is described as a set of SystemC functional modules interacting together through the interface provided by abstract middleware (i.e., AME_3). Simulation mechanisms are provided by the SystemC simulation environment in which each function and the middleware are considered as concurrent processes.

At the second stage, system/network partitioning has been performed and the *system model* of each node interacts with other nodes through communication links described in the *network model*. At this level a different version of the abstract middleware library (i.e., AME_2) is used; it provides the same interface to the application code but communications are implemented through packet exchanges. System models are simulated by the SystemC simulator while packet delivery is simulated by a network simulator. A cooperation between the two simulation environments is needed; in particular, the network simulator must provide SystemC with an API to transfer packets from system models to the network and viceversa; for this purpose some general co-simulation approaches have been implemented as described in Section 4.1. On the other side, AME_2 has to provide a network simulator interface to establish a connection with the network simulator; this interface is presented in Section 4.2.

At the third stage, *HW/SW partitioning* is applied to each node to map functionalities to HW and SW components according to several constraints (e.g., performance, cost, and component availability). At this level communications APIs provided by AME_1 services are the same API of the previous step (AME_2). HW components are involved in the simulation through HW/SW/network simulators (as described in Section 4.1); in this case the application code can be changed for the modelling of the HW components.

4.1 HW/SW/Network simulation

In this Section we split the problem of HW/SW/Network co-simulation into System/Network co-simulation, where systems represent processing nodes made up of HW and SW components, and HW/SW co-simulation, where the interactions between HW and SW components within each node are considered.

Efficient modelling and simulation of networked systems require that tools exhibit a good level of scalability, completeness, fidelity, and reusability. The simulator should be able to handle large networks of thousands of nodes in a wide range of configurations (scalability). It should be able to cover as many system interactions as possible, accurately capturing behaviour at a wide range of levels (completeness) and revealing unanticipated interactions, not just those a developer suspects (fidelity). Finally, the simulator should bridge the gap between algorithm and implementation, allowing developers to test and verify the code that will run on actual hardware (reusability). Different aspects should be addressed during the modelling and simulation of networked embedded systems. They can be classified according to three domains, i.e., software, hardware, and network.

4.1.1 Software

The characteristics to simulate in the software domain are: the functional and timing behaviour of the software and its interaction with external events through interrupts (e.g., the presence of concurrency issues). While the functional behaviour of a system can be easily simulated through general-purpose languages such as C or C++, other characteristics can be reproduced only by a cycle-accurate emulation of the CPU through an instruction set simulator [50] and the support of debug facilities. The instruction set simulator (ISS) is an application which runs on a host workstation and executes programs written and compiled for a different processor (target platform). ISS simulates the behaviour of a program and the associated operating system at the instruction-set level; simulation is cycle-accurate, i.e., the number of simulated instruction cycles to perform a given operation is the same as on actual hardware. This tool can be used to verify the interactions between the application and the operating system and, if a power model of the CPU is available, to evaluate power consumption. Using this tool, developers can test and verify the same object code that will run on actual hardware. Simulations performed by ISS lack realistic timing information since instruction cycles, not seconds, are the basic time unit. For this reason, this tool cannot be used to model asynchronous events triggered by hardware components or by the network.

4.1.2 Hardware

Also in this domain, the functional behaviour of the system should be reproduced at the first design stage. Then, the tool should allow to refine the description to represent the architecture as a set of interconnected blocks (structural view). In this flow, non-functional information should be managed, e.g., timing behaviour, area utilization and power consumption. A desired feature for a HW simulation tool is its support for the synthesis of the architecture. A traditional language for

hardware description is VHDL while SystemC [45] is gaining increasing attention for its great flexibility in describing devices at different abstraction levels, from system level down to RTL and gate levels. SystemC is a C++ class library that provides the constructs required to model system architectures including hardware timing, concurrency and reactive behaviour that are missing in standard C++. In literature SystemC was already used to describe network-on-chip architectures [51] and to simulate the lowest network layers of the Bluetooth communication standard [52].

4.1.3 Network

Network can be modelled at different levels of detail, from packet level down to the electromagnetic propagation. Simulated values can be either generated by an analytic model or taken from experimental data sets; the first approach is more general but it strongly depends on the model validity and may be computational intensive. Network Simulator, NS-2 [53], is the most widely used discrete event simulator for computer networks. It is written in C++ and provides modules for the simulation of well-known protocols both wired and wireless. NS-2 simulates networks at the packet level and provides facilities to collect statistics at different detail levels. Some extensions have been developed to simulate sensor networks for environmental monitoring applications [54–58]. The main weakness of NS-2 is that it does not model concurrent processes within the network node. With NS-2, simulation scenarios are created by connecting together different kind of objects, i.e., nodes, agents and applications describing different layers of the ISO/OSI model. Since a cross-layer approach is preferred in the design of wireless sensor networks, NS-2 should be deeply modified to exploit the interaction between protocols and applications. Besides, implementing a new protocol requires the update of a lot of NS-2 configuration files. Some specific tools were developed in the past for the simulation of wireless sensor networks (e.g., TOSSIM [47], AVRORA, EMSTAR, ATEMU, SQUALNET) even if most of them are targeted to a specific architecture (e.g., Berkeley's motes). Using different tools for network modelling and node implementation limits the reuse of code and test-benches. Although this issue can be tolerated in today's wireless systems often designed using off-the-shelf hardware components, the high integration of next-generation networked embedded systems could require that hardware design and network simulation will be applied on the same models.

4.1.4 System/Network co-simulation

The design of networked embedded systems (NES) requires the availability of both a traditional system-level modelling tool and a networking modelling environment. Furthermore, simulation results from these two domains should be merged to provide a comprehensive view of the System and the Network. Accounting for the presence of a communication network at the early stages of the design flow is essential for several reasons. First, it allows verifying the embedded system in a realistic scenario to assess that functional requirements and design constraints are

met. In particular, reproducing the network behaviour is crucial to verify the communication constraints (e.g., throughput and delay). Second, the same network environment can be used to validate successive refinements of the embedded system. Third, the implementation of the communication structure of the embedded system can be validated through the successful interoperation with the protocol stack described at higher abstraction levels by the network modelling tool. Finally, when applicable, network protocols can be seen as an additional "design variable" for further optimization of the whole architecture.

In [60–63] SystemC has been used to model HW and SW parts of the system while NS-2 has been used to model the external network. Some changes have been done to the simulator kernels to perform a synchronized simulation. However, some points have not been clarified:

- the connection of system components at different ISO/OSI network layers,
- the decision of which components should be modelled in SystemC and which in the network simulator,
- the integration of the TLM approach in this HW/SW/Network design flow.

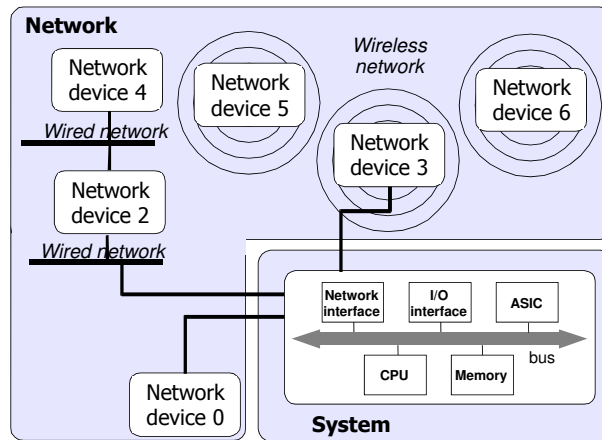


Fig. 4.1. Example of networked embedded system: Internet router.

Figure 4.1 shows a simple representation of a networked embedded system and highlights the presence of system and network parts. The System usually consists of a CPU, a memory to store application code and data, one or more network interfaces, I/O interfaces for data acquisition and user interaction, and other components –ASIC’s or FPGA’s– designed to efficiently perform specific functions [64]. An application-specific SW is deployed over this HW architecture often together with an operating system which bridges HW and SW components. Among the system modelling languages, SystemC [45] is gaining increasing attention for its great flexibility in describing devices at different abstraction levels and for its interoperability with other languages, e.g., VHDL. The Network consists of a set of

nodes connected to the System through communication links. Different parameters can be defined for each link, e.g., the type of channel (wired/wireless), the bandwidth, and the delay. The way in which nodes exchange data represents the protocol specification. Even if general purpose languages can be used to reproduce the behaviour of network protocols, specific network simulators, e.g., NS-2 [53], already provide models for well-known network protocols, e.g., Ethernet, WiFi, and TCP/IP.

System modelling

Figure 4.2 shows the system model of an Internet router; the model follows a modular approach where the main router functionalities are identified:

- the management process,
- the TCP protocol implementation to handle remote connections (e.g., to support a remote shell),
- packet integrity checking and forwarding.

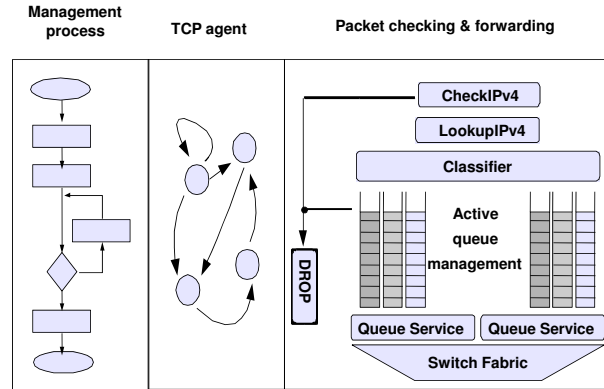


Fig. 4.2. System model of a networked embedded system: Internet router.

In the first design stage, the functional behaviour of the system should be reproduced; at this level it is not yet determined which functionalities will be implemented in HW and which in SW. Thus, different modelling approaches are allowed at this level, e.g., an operation sequence to describe the management process and a finite state machine to specify the TCP connection management. For a given functionality, e.g., in the Figure, the packet checking and forwarding, the detail level of the description can be increased to represent its architecture as a set of interconnected blocks (structural view).

Among the system modeling languages, SystemC is gaining increasing attention for its great flexibility in describing devices at different abstraction levels, from

system level down to RTL and gate levels. SystemC is a C++ class library that provides the constructs required to model system architectures including hardware timing, concurrency and reactive behavior that are missing in standard C++. SystemC supports Transaction Level Modeling (TLM) [46] which aims at standardizing the refinement process of the System model to enable re-use between abstraction levels within the same project and between projects belonging to different manufacturers.

Network modelling

Network can be modelled at different levels of detail, from packet transmission down to signal propagation. Network simulators reproduce the functional behaviour of protocols, manage time information about transmission and reception events and simulate packet losses due to congestion or link failure. Network Simulator, NS-2, is the most widely used discrete event simulator for computer networks. It is implemented in C++ and its source code is open. It is widely used in many research activities because it includes modules for the simulation of well-known protocols both wired and wireless [22,23]. The modelling approach of NS-2 follows the well-known ISO/OSI reference model, i.e., a layered architecture in which each layer provides services to the upper layer and uses services of the lower layer.

Figure 4.3 shows the example of the Internet router completely described by using NS-2 entities; this approach allows to model the surrounding network; to improve clarity a dashed bold box separates router components from the surrounding network. Round entities are called *Nodes*; they are connected together by *Links* which can be wired (continuous lines) or wireless as in case of Node 3, 5, and 6. Nodes and links reproduce the lowest three ISO/OSI layers and, in particular, Node 1 reproduces the router behavior for what concerns packet integrity checking and forwarding; it is worth noting that packet integrity checking requires a bit-level simulation which is not currently supported by NS-2. Square entities are called *Agents* and represent the Transport Layer; they are attached to nodes and connected together to reproduce end-to-end UDP or TCP connections (continuous lines with arrows). Agent TCP3 reproduces the TCP agent implemented in the operating system of the router. Rhomboidal entities are called *Applications* and represent the Application Layer; they reproduce application sessions (dotted arrows) which inject packets into the network through Agents according to traffic models derived from either actual applications or statistical functions. Application `MGM server` reproduces the router management process. *Nodes*, *Agents* and *Applications* are connected together by object references in the network simulator (dotted lines).

The set of entities (`web client`, `TCP1`, `TCP4`, `web server`) represents an HTTP session build upon a TCP connection whose packets flow through Nodes 0, 1, 2, and 4. Such packets cross the router which forwards them according to the routing table. The set of entities (`CBR source`, `UDP1`, `UDP2`) represents a constant-bit-rate flow of UDP packets (e.g., a Voice-over-IP conversation). According to the routing rules, packets are delivered through the wired link between Node 1 and 3 and the wireless link between Node 3 and 6. The set of entities (`MGM client`, `TCP2`, `TCP3`, `MGM server`) represents a remote management session build upon a TCP

connection. It is worth to note that the router can be considered as an intermediate system for what concerns packet checking and forwarding and as an end system for what concerns TCP and management services.

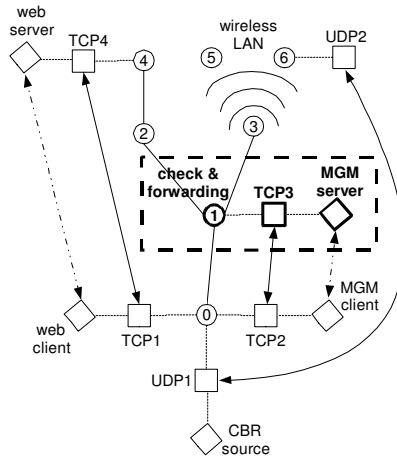


Fig. 4.3. NS-2 model of an Internet router connected to a simple network.

When designing an integrated simulation framework, two are the main issues that affect the integration:

- system/network co-simulation partitioning: the designer should decide which functionalities have to be modeled by the System tool and which by the Network tool; this decision should be driven by methodological criteria;
- the network level of interaction between NS-2 entities and the SystemC modules; in fact specific extensions of the NS-2 entities could allow the interaction between the corresponding SystemC modules at different levels of the ISO/OSI stack.

System/Network co-simulation partitioning

This Section deals with the subdivision of functionalities that should be modelled in the System and in the Network. For example, in Figure 4.3 both the router and the surrounding network have been modelled by using the network simulator but during the design of this embedded system some of its functionalities (e.g., packet checking and forwarding, TCP agent, management service) should be extracted from the network model and described in a suitable system design language. The general criterion is that functionality to be implemented in the embedded system has to be described as component of the System model instead of the Network model. Conversely, the network modelling tool should be used to describe that functionality which is not part of the design process but of the environment in which the embedded system will operate. In this context, the use of third-party

components provided by the network modelling tool not only speeds up the design and simulation process, but also contributes to validate the designed communication structure by testing its interaction with an abstract reference protocol specification shared by the research community.

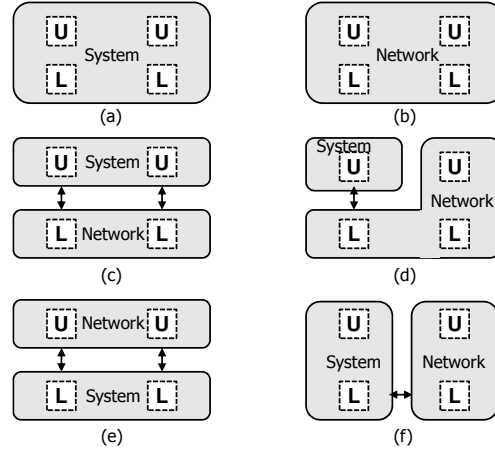


Fig. 4.4. Different possible approaches for System/Network co-simulation partitioning.

Different approaches for System/Network partitioning are shown in Figure 4.4. Two end systems are represented by the corresponding protocol stacks; for simplicity's sake each stack is reduced to an upper layer and a lower layer represented by boxes labelled with U and L respectively. Each approach has its own advantages and drawbacks which should drive its adoption. In Figure 4.4.a the SystemC simulator is used not only to model components but also to simulate network communications. In this case all TLM levels are supported since the model of the network can be both timed and un-timed; the main drawback of this approach is that all network protocols should be re-implemented by the designer. In Figure 4.4.b only the network simulator is used as in the example reported in Figure 4.3. Network modelling tools lack fidelity since they describe functionality without reproducing the interaction among different components within the single node as in actual systems; this fact limits the reusability of the functional description in the next design phases. Instead, system modelling tools have the advantage that the model can be refined, i.e., transformed from a behavioural to a structural description, and traditional validation techniques can be applied to it.

In the other cases both simulators, SystemC and NS-2, are used and arrows represent interactions between them. The common drawback of these approaches is that the required synchronization increases simulation time. In Figure 4.4.c the upper layer of the NES is the focus of the design process and thus it is modelled by SystemC. The interaction of two instances of this component takes place through the corresponding lower layer entities modelled by the network simulator. The designer takes advantage by the use of a specific tool for network simulation con-

sidering that such lower entities are outside the design scope. Also in Figure 4.4.d the focus of the design process is the upper layer of the NES but, for validation purpose, a SystemC instance of the component interacts with a peer entity modelled by the abstract protocol specification of the network simulator. In this case, the generation of test patterns is simplified by the use of a specific tool for the reproduction of network behaviour. In Figure 4.4.e the focus of the design process is the lower layer and therefore it is modelled by SystemC which also reproduces the communication channel. The upper layer is modelled by the network simulator thus simplifying the generation of test application models which are outside the design scope. Finally, in Figure 4.4.f both the upper and lower layers belong to the design process and thus they are modelled by SystemC and validated by the reference stack specification of the network simulator.

NS-2 simulations always consist of a sequence of timed events (e.g., start and stop of packet flows or packet transmission and reception). SystemC models can be either timed or un-timed. For this reason a kind of adapter is needed in the SystemC model to connect un-timed SystemC components to an NS-2 topology.

System/network interaction

In previous works [60, 62] interactions between system and network models took place only at the transport layer preventing the use of SystemC to model components at the lower ISO/OSI layers. Figure 4.5 presents a possible co-simulation approach for the example of Figure 4.3. The functionalities highlighted in the SystemC implementation of the router belong to different ISO/OSI layers and should exchange information with the corresponding entities in NS-2. For example, the SystemC module implementing the forwarding functionality should receive from the network a layer 3 packet reporting the destination node address.

For this reason, three new entities have been added to the NS-2: the `ns_sc_link`, the `ns_sc_agent`, and the `ns_sc_app`; in Figure 4.5 they are represented by the filled shapes.

The `ns_sc_link` connects NS-2 *Nodes* with SystemC modules whose functionalities belong to the lowest three ISO/OSI layers. This kind of entity conveys network addresses and bit-accurate packet descriptions. The `ns_sc_agent` connects NS-2 *Agents* with SystemC modules whose functionalities belong to the transport layer. This entity conveys transport addresses and acknowledgements (in case of TCP connections). The `ns_sc_app` connects NS-2 *Applications* with SystemC modules whose functionalities belong to the application layer. This entity conveys the application content of network transmissions.

In NS-2 the *Link* simply reproduces a transmission channel, while the *Node* is a more complex entity which also contains the rules for channel access; therefore, when a new kind of network is introduced in NS-2 (e.g., wireless networks), the *Node* should be extended while the *Link* remains unchanged. For this reason, our modified version of the *Link* entity is compatible with the future releases of NS-2. It is worth noting that all the reported interactions involve event and data exchange between SystemC and NS-2.

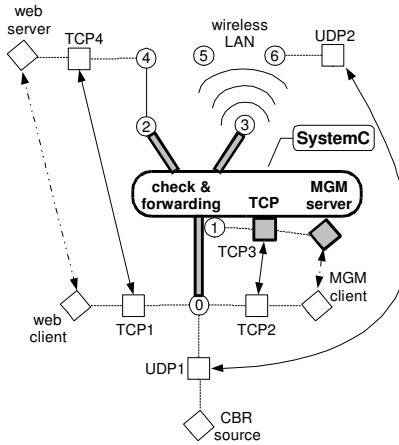


Fig. 4.5. System/Network co-simulation applied to the design of an Internet router.

Co-simulation implementation

In this Section we describe the implementation details of our System/Network co-simulation framework. A message structure was defined to transfer data between the simulation kernels. An addressing scheme has been introduced to identify the destination of a packet arriving from the other simulator; each instance of the SystemC (`ns_in` and `ns_out`) have a unique identifier corresponding to the NS-2 output and input entity, respectively; each instance of the NS-2 entities (`ns_sc_link`, `ns_sc_agent`, and `ns_sc_app`) has two unique identifiers (one for each data direction). In the following we will describe:

- the syntax of the SystemC `ns_in` and `ns_out` ports;
- the syntax of the new NS-2 entities;
- the message format for data transfers between the simulation kernels;
- the implementation of kernel synchronization.

Syntax of the new SystemC ports

From the SystemC side, the new ports `ns_in` and `ns_out` have been added to allow the user to send/receive a packet to/from a NS-2 object. They are derived by template classes `sc_in` and `sc_out`, and are managed by overridden methods `read()` and `write()`. These ports implement the concept of co-simulation external ports reported in [65]. Figure 4.6 shows the declaration of a SystemC module communicating with the NS-2 entities described in Figure 4.7.b. Each port allows data transfer in only one direction and, for this reason, only one identifier is associated to it during instantiation.

Methods `write()` and `read()` are used to send (read) a packet to (from) the corresponding NS-2 entity as shown in the SystemC code of Figure 4.7.a.

```

SC_MODULE(module) {
  ns_in *agent_in; // port to receive a packet
                  // from the NS-2 agent
  ns_out *agent_out; // port to send a packet to
                   // the NS-2 agent
  ns_in *link_in; // port to receive a packet
                 // from the NS-2 link
  ns_out *link_out; // port to send a packet to
                   // the NS-2 link
  ns_in *app_in; // port to receive a packet
                // from the NS-2 application
  ns_out *app_out; // port to send a packet to
                  // the NS-2 application

  // functions called when a packet arrives from NS-2

  void agent_proc();
  void link_proc();
  void app_proc();

  SC_CTOR(module) {
    // instantiation and address assignment of
    // input ports
    agent_in = new ns_in(10);
    link_in = new ns_in(11);
    app_in = new ns_in(12);

    // assignment of callbacks to input ports
    NS_PROC(agent_proc,agent_in);
    NS_PROC(link_proc,link_in);
    NS_PROC(app_proc,app_in);

    // instantiation and address assignment of
    // output ports
    agent_out = new ns_out(20);
    link_out = new ns_out(21);
    app_out = new ns_out(22);
  }
}

```

Fig. 4.6. Declaration of a SystemC module communicating with an NS-2 model through ns in and ns out ports.

Whenever the SystemC kernel receives a data message from NS-2, it generates an event on the `ns_in` port to which the data message is destined and with the timestamp specified in the message. The kernel then wakes up the function which has been assigned to that event as in traditional inter-module communications. Whenever a SystemC process writes a packet on a `ns_out` port, the `write()` method stores it into the *DataToSend Queue*. When the simulation control passes to the kernel it builds a message with such packet according to the algorithm reported in Figure 4.9.

Syntax of the NS-2 entities

New classes have been created to implement the `ns_sc_agent`, the `ns_sc_app` and the `ns_sc_link`. In traditional NS-2 Agents and Links, the `recv()` method is called when a packet arrives from another entity of the network model. This method has been modified in case of the `ns_sc_agent` and `ns_sc_link` to send the packet to the Scheduler which put it on the *DataToSend Queue* together with the SystemC destination address. The elements of this queue will be transmitted to the SystemC kernel. A similar approach has been followed for the `ns_sc_app` but in this case both the `recv()` method and the `process_data` method have been extended.

All the packets coming from SystemC are embedded in messages which arrive to the NS-2 kernel. For each new class, the `cosim_recv()` has been added to process

```

1 set n0 [$ns node]
2 set n1 [$ns node]
3 $ns duplex-link $n0 $n1 10Mb 20ms DropTail
4 set agent0 [new Agent/ns_sc_agent 20 10]
5 $ns attach-agent $n0 $agent0
6 $ns ns_sc_link $n1 21 11 1Mb 10ms DropTail
7 set agent1 [new Agent/UDP]
8 $ns attach-agent $n1 $agent1
9 set app0 [new Application/ns_sc_app 22 12]
10 $app0 attach-agent $agent1

```

(a)

```

void module::agent_proc() {
    ...
    Packet *p;
    p = ... // the packet is explicitly built
    // sending packet
    agent_out.write(p, sizeof(Packet), receiver);
    // reading packet
    agent_in.read(p, sizeof(Packet));
    ...
}

```

(b)

Fig. 4.7. (a) SystemC code with read and write operations through ns in and ns out ports. (b) Portion of TCL script showing the use of ns_sc_link, ns_sc_agent, and ns_sc_app.

incoming data. Therefore, the kernel invokes this method on the entity which is the destination of the data contained in the message.

Figure 4.7.b reports an ideal portion of TCL script containing all the new entities. The first three lines create two nodes connected through a link. Line 4 creates an instance of the ns sc agent; the last two numerical parameters are used to identify the output and input port of SystemC, respectively. This agent is placed upon the first node (Line 5). Line 6 creates an instance of the ns_sc_link between the second node and a SystemC model (whose output and input ports are identified by the first two numerical parameters). The instance of ns_sc_agent is attached to an UDP agent on the second node (Lines 7-8). Line 9 creates an instance of the ns_sc_app which is connected to a SystemC model (whose output and input port are identified by the two numerical parameters). This entity is attached to the UDP agent (Line 10).

Message structure

The information exchanged between the simulators are organized into messages following the structure shown in Figure 4.8. The Type field indicates if the message aims at basic time synchronization or involves data exchange; in the latter case this field is followed by a data array reporting each data event and the corresponding destination entity (note that more data events may be generated with the same

timestamp); the next two fields report the current timestamp and the timestamp of the next simulation event, respectively. Message creation is performed by the simulation kernels and it is transparent to the designer.

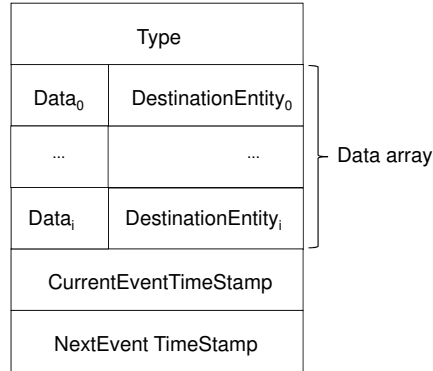


Fig. 4.8. Structure of the messages exchanged by the simulation kernels.

Kernel extension for synchronization

Figure 4.9 shows the pseudocode of the synchronization procedure, which is executed by both kernels. The subscript $k1$ refers to the kernel currently executing the code, and $k2$ to the other kernel; both kernels are modified so as to incorporate this procedure.

After a common setup phase (Line 1), one of the two kernels sends the timestamp of the next event to the other kernel (Line 2). Then the main synchronization loop (Lines 3-22) evolves around the reception of messages on the channel that links the two simulators (Line 4). Anytime a message is received, it is first checked if it is a `Data` message type; in this case, it causes the invocation of the `cosim_recv()` method (Lines 6-7) of the SystemC or NS-2 receiver.

Then, the timestamp T_{k2} of the remote event is extracted and compared to the timestamps T_{k1} of the local events (Lines 8-9). The loop of Lines 10-18 manages the processing of all the events in the local queue that are lagging behind the remote simulation time T_{k2} . If the generic event E_{k1} implies the transmission of data (Line 13), the current message M_{k1} for time T_{k1} is properly setup by specifying the corresponding data field D_{k1} and remote recipient j (Line 14).

This condition is flagged (Line 15) for later use. For any new value of T_{k1} (yet still $\leq T_{k2}$), a new message M_{k1} is actually allocated (this operation is abstracted away in the pseudocode). The do while loop at Lines 11-18 allows the dispatching of all events with the same scheduling time. When the loop at Lines 10-18 exits it is time to send a message to the other kernel. If the flagged condition at Line 19 is

```

SynchronizedScheduler() {
1  Setup phase
2  Send(Mk1) // done by only one of
           // the two kernels
3  do {
4      Receive(Mk2); // from the other kernel
5      state = TRUE;
6      if(Mk2.Type == Data) { // a data msg
7          for each Mk2.DestinationEntityj {
           Call Entity.cosim_recv();
           }
           }
           Tk2 = Mk2.NextEventTimeStamp;
8      Get next event Ek1 from ReadyQueue;
9      Tk1 = TimeStamp(Ek1);
10     while(Tk1 ≤ Tk2 && state) {
11         do { // events that are lagging behind
12             dispatch event Ek1;
13             if(Ek1 requires sending data to entity j){
14                 DataToSend.enqueue(Dk1, j);
15                 state = FALSE;
           }
           }
           Get next event Ek1 from ReadyQueue;
           Tk1_olD = Tk1;
           Tk1 = TimeStamp(Ek1);
17     } while(Tk1 == Tk1_olD);
18 }
19 Allocate a new message Mk1;
   if(state) {
       Mk1.Type = Data;
       Mk1.CurrentEventTimeStamp = Tk1_olD;
       Mk1.dataArray = DataToSend();
       state = TRUE;
   } else Mk1.Type = Time;
20 Mk1.NextEventTimeStamp = Tk1;
21 Send(Mk1);
22 } while(there are messages);
}

```

Fig. 4.9. Structure of the messages exchanged by the simulation kernels.

true, the kernel sets up a data message, otherwise a time message is set up. When the message is ready it is sent to the other kernel.

4.1.5 HW/SW (SystemC/ISS) co-simulation

Related works

The concept of HW/SW co-simulation regards the integrated simulation of both HW and SW components of the system under test.

Concerning SW simulation some solutions are based on ISS-free schemes. For example, a delay-annotated software simulation tool is described in [66]. This approach determines the timing behavior of the SW at level of each C++. Although this approach seems to be a fast and easy alternative to a full ISS, it lacks detailed information on how to determine the delay of each C++ statement, since processor manufacturers only provide delay information for Assembly instructions and not for C++ statements; moreover, it does not consider compiler optimizations. Delay annotation has been improved by considering the Assembly instructions generated by the actual compiler for the target platform [67]. In the same work delay annotation has been also combined with native execution of target code on the host platform.

The presence of an ISS increases accuracy since this tool is specifically designed to emulate the target CPU and the executed SW is the same which run on the actual platform. Furthermore modularity is enforced since a change in the CPU

type implies only the recompilation of the SW part; ISS-free schemes, conversely, would require complete re-targeting of the software. In case of external ISS, the simulation performance could be limited by the communication overhead between the HW simulator and the ISS. A possible way to reduce this overhead consists in reducing message exchanges through dynamic prediction of transaction occurrence time for both software and hardware models [68].

Concerning the simulation of the HW part, a possible solution is the creation of virtual prototypes through the use of HW description languages, such as VHDL, Verilog and SystemC. Such virtual prototypes are available at the early stage of the design flow and the designer can use different abstraction levels trading off between accuracy and speed. Another solution emulates HW by using inexpensive FPGA [69]; communication between the SW simulator and the FPGA occurs via a flexible interface based on shared communication registers. This solution requires the maintenance of multiple code bases of the design i.e., one for the FPGA-based prototype, and one for the real HW. Furthermore, the HW description has to be at RT level to be used on the FPGA. Simulation with actual HW prototypes is another solution even if it is available late in the design flow [61].

Concerning the simulation of multi-CPU systems we have to consider tools for the simulation of MPSoC such as Simics [70] and MPARM [71], and tools for wireless sensor networks such as TOSSIM [47].

Simics is a commercial full-system simulator. It provides a virtual version of a target hardware on top of a standard host PC. Even if modelling of new HW components is possible, this product is mainly devoted to HW integrators and software developers since the virtual hardware runs the same binary software as the physical target system. HW models are described at behavioral level and therefore they cannot be inserted into a traditional HW design flow.

MPARM is a SystemC-based modelling and simulation environment for MP-SoC; it includes models for processors, the AMBA bus architecture, memory models and support for parallel programming. A fully operating linux version for embedded systems has been ported on this platform, and a cross-toolchain has been developed as well.

TOSSIM is a simulation tool for wireless sensor nodes running TinyOS operating system. This tool reproduces the behavior of the actual application code by modeling TinyOS services on the host machine. TOSSIM is strictly targeted to this specific architecture, it only supports networks of homogeneous nodes and it does not offer a direct path to hardware design and synthesis.

Co-simulation implementation

Starting from the analysis of past literature we can conclude that SystemC and ISS give the best results in terms of flexibility and availability at the early stages of design flow. SystemC can be used to model HW components at different abstraction levels while ISS can accurately emulate the behavior of the target CPU. Even if, in the past, communication overhead between the tools was considered a limiting aspect, we expect that the power of today's multi-core systems and an accurate tuning of inter-process communications might overcome this drawback and make SystemC/ISS co-simulation a good solution.

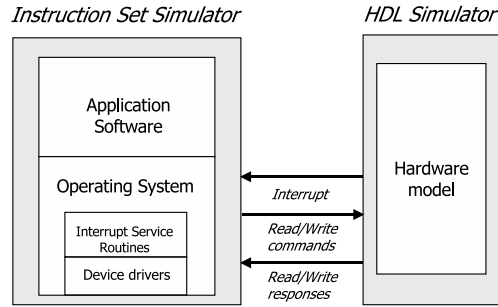


Fig. 4.10. HW/SW co-simulation architecture.

The starting point of our work is the un-timed SystemC/ISS co-simulation methodology described in [72] and depicted in Figure 4.10. This co-simulation model consists of an ISS and a simulator of HW models (e.g., SystemC), both executed as processes in the host operating system and connected together through an inter-process communication (IPC) channel. The ISS executes the target SW (i.e., applications, OS, and device drivers); read and write accesses to the HW components (either memory-mapped or I/O-mapped) are transmitted to the HW simulator through read/write commands while HW interrupts are reported to the ISS through corresponding messages in the opposite direction. The generation and management of such messages is transparent to the target SW since the ISS intercepts interrupt and read/write Assembly instructions which involve HW registers outside the CPU.

The advantages of this solution are:

- Actual device drivers for the target system can be used during the simulation provided that the addressing schema of HW devices is the same.
- Communication between SW and HW parts is described in terms of abstract messages instead of true bus cycles.
- Communication between SW and HW parts is managed by simulation kernels to gain efficiency and to be completely transparent to the designer.

The aim of this work is the extension of this methodology to support:

- timing-accurate synchronization between HW and SW models;
- coexistence of different execution environments, one for each processing unit;
- clustering of HW and SW models to represent each processing unit.

Time synchronization

This section addresses the extensions for timing accuracy added to the framework previously described. A mandatory requirement to implement the timing-accurate ISS/SystemC co-simulation is the notion of time inside the ISS. With this feature the local time of the ISS can be compared with the time of the SystemC simulator.

The synchronization mechanism follows an asymmetric scheme, where one of the two simulators (the master) explicitly controls the execution of the other (the

slave). Time synchronization is provided by adding time information in the messages exchanged between the SystemC simulator and the ISS. Both simulator kernels have been modified for this purpose. In this way synchronization is more efficient and the modeler does not have to know its details.

```

1  create_IPC_channel(ISS);
2  do {
3      if ( !channel->isEmpty() ) {
4          receive(msg);
5          if (SystemC_Time < msg.ISS_Time)
6              add_timed_event(<operation,
7                              msg.ISS_Time, ISS_Port>);
8          else
9              send_response_to_ISS();
10     }
11     event = extract_event_from_queue()
12     if (event == "ISS_TIMED_EVENT")
13         send_response_to_ISS();
14     else
15         // NORMAL SystemC Kernel code
16     } while (...SystemC events...);

```

Fig. 4.11. SystemC procedure for time synchronization with ISS.

Figure 4.11 shows the pseudocode of the time synchronization procedure in the SystemC kernel, representing the co-simulation master. The SystemC co-simulation kernel starts by creating the IPC channel toward the ISS (Line 1). Then, simulation loop starts and SystemC checks (Lines 3-4) for ISS requests (i.e., read/write registers commands); each command also reports the time (in the time space of ISS simulation) in which it has been generated (*ISS_Time*). SystemC compares *ISS_Time* with local time (*SystemC_Time*) If local time is smaller than the ISS time, then the SystemC simulator is lagging behind (Line 5); in this case the SystemC inserts the request as a new event into the event queue (Line 6); that event will be scheduled in the future when ISS has generated the request (i.e., *ISS_Time*). Conversely, if SystemC simulator is ahead of time with respect to the ISS simulator (Line 7), it executes immediately the read/write operation and sends the response to the ISS (Line 8). Then, the traditional event-driven scheduling is performed (Line 10); if the scheduled event has been added during the previous phase (i.e., it is an *ISS_TIMED_EVENT*), the simulator executes the corresponding read/write operation and sends the response to the ISS (Line 12); otherwise, the SystemC simulator executes the SystemC event as normally (Line 14). The above operations are repeated for the whole simulation (Line 15).

At the other side, ISS simulation kernel has to be modified so that messages exchanged with the master simulator contain time information.

A possible weakness of this mechanism regards the case in which SystemC simulator is ahead of time with respect to the ISS simulator; in this case the result of a read operation or the behavior of SystemC after a write operation may depend on the delay of ISS with respect to SystemC. A possible solution consists in forcing periodical synchronization messages between the tools thus increasing significantly the communication overhead. Another solution, at least in case of read operations,

could be the generation of a warning message if a HW register is updated twice by SystemC without being read by the ISS.

Multi-CPU co-simulation

This section presents the SystemC simulator extensions to support a multi-ISS simulation. The multi-ISS co-simulation mechanism is based on the communication protocol established between the ISS and SystemC as described in Section 4.1.5.

The communication between the HW simulator and the ISS is implemented by means of three new type of ports added to the SystemC library, i.e., `iss_in`, `iss_out` and `iss_interrupt`. The `iss_in` port is derived from the standard `sc_in` port and it is used to read data coming from ISS, the `iss_out` port extends the SystemC `sc_out` port and it allows to send data from SystemC to the ISS. These special ports can be used to model HW registers, thus allowing the ISS to read and write them. In the following text, we assume that HW registers are memory-mapped. The connection between the two sides of the co-simulation is performed by binding specific addresses of the ISS memory space to SystemC `iss_in` and `iss_out` ports contained in the HW models. When the CPU accesses some registers through their addresses, the SystemC kernel determines the corresponding special ports of the corresponding hardware model. The link between the SystemC port and the corresponding memory address in the ISS is implemented by using a binding table stored in the SystemC kernel.

To support multi-ISS simulation, the SystemC simulator has been modified to manage messages incoming from all ISS's. First of all, the constructor of previously described special ports (i.e., `iss_in` and `iss_out`) has been extended to obtain a string which identifies the ISS instance which can issue read/write requests. For example, an `iss_in` port has to be initialized as follows:

```
iss_int * reg1 = iss_in(0x12340000, ISS1)
```

Corresponding to this declaration, the following record is inserted into the binding table:

```
< reg1, 0x12340000, ISS1 >
```

This mechanism allows to cluster HW models and ISS instances to model independent processing units containing a CPU and some related HW devices.

The SystemC kernel creates an IPC channel for each ISS, as shown in Figure 4.12. Since the SystemC kernel has to know the mapping between the ISS identification string and the IPC channel, during the setup phase, each ISS instance sends its identification string to SystemC which updates the binding table accordingly. An example of binding table is depicted in Figure 4.13; it shows the relationship between addresses, ISS identification strings and IPC channels. It is worth noting that the same address can be used for different registers connected to different ISS's since their memory spaces are disjoint.

Figure 4.14 shows the pseudo-code of the SystemC simulator to support multi-ISS simulation; bold text represents added code with respect to the pseudo-code described in Figure 4.11. Lines 1-4 implement the setup phase. After this phase,

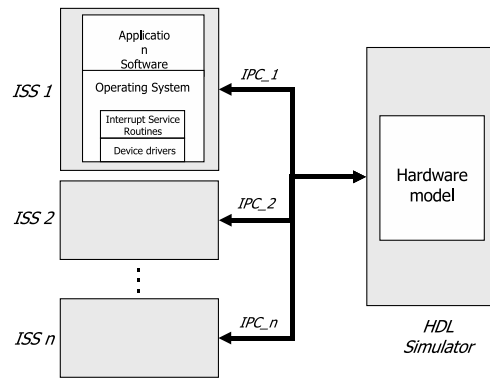


Fig. 4.12. General architecture of the multi-ISS co-simulation.

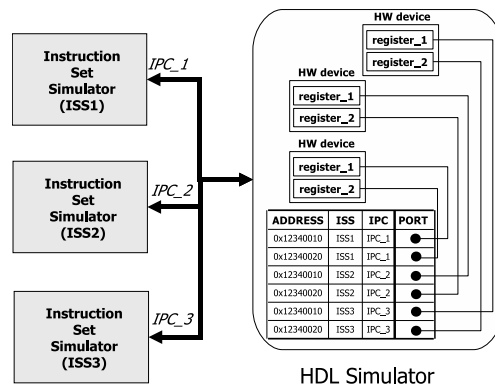


Fig. 4.13. The communication between different ISS instances and SystemC.

each ISS is able to send/receive data to/from the SystemC simulator. In the multi-ISS scenario the SystemC kernel could receive messages from different ISS's. Therefore each IPC channel has to be monitored to verify the presence of an ISS request (Line 6 and 7). Corresponding to each ISS request, the SystemC kernel has to reply to the proper ISS identified by the `iss_id` variable. Finally, the simulator extracts the next event from the queue to schedule it. In the multi-ISS case, SystemC has to find the ISS able to receive the response (Line 17).

4.1.6 HW/SW (SystemC/QEmu) co-simulation

In the design of ever complex embedded systems, a major task is handling several platforms consisting of different processors and operating systems as well as a large amount of HW devices such as memory, DSPs, I/O interfaces and ASICs. As described in Section 4.1.5, Instruction set simulators (ISS's) can be used to reproduce the behavior of target processors; their use offers several advantages, such as the flexibility of specifying different targets and the wide availability of standard development tools [79]. Even if some ISS has the capability to model simple HW components, such as memory and timers, in general, the simulation of

```

1  for (iss_id=0; iss_id < ISS_NUM; iss_id++) {
2  create_IPC_channel(iss_id);
3  update_binding_table(iss_id);
4  }
5  do {
6  for (iss_id=0; iss_id < ISS_NUM; iss_id++) {
7  if ( !channel[iss_id]->isEmpty() ) {
8  receive(msg);
9  if (SystemC_Time < msg.ISS_Time)
10     add_timed_event(<operation,
11                    msg.ISS_Time, ISS_Port, iss_id>;
12     else
13     send_response_to_ISS(iss_id);
14  }
15  event = extract_event_from_queue();
16  if (event == "TIMED_EVENT")
17     iss_response = find_iss_by_event(event);
18     send_response_to_ISS(iss_response);
19  else
20     // NORMAL SystemC Kernel code
21  } while(...SystemC events...);

```

Fig. 4.14. SystemC procedure to support multi-ISS co-simulation.

HW components relies on the use of HW description languages and their simulation environments.

Figure 4.15 depicts different co-simulation strategies. Different strategies can be used to simulate HW components:

- *host-mapping approach*: devices are mapped on the actual ones on the host machine;
- *model-level co-simulation approach*: devices are simulated by using HDL models and every driver controls the corresponding device through a dedicated channel connected to the corresponding HDL model;
- *tool-level co-simulation approach*: devices are simulated by using HDL models and synchronization between HW and SW simulations is done at tool level by exchanging messages through a single control channel.

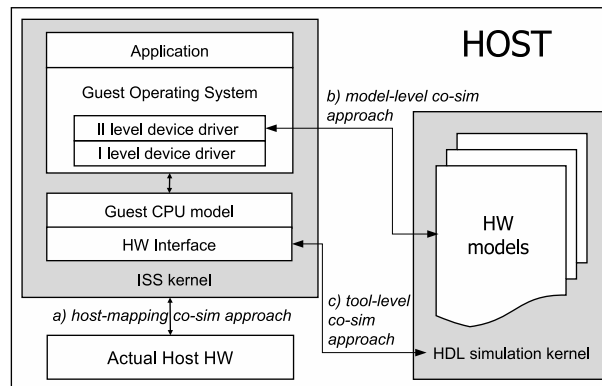


Fig. 4.15. HW/SW Co-simulation strategies.

In literature the model-level co-simulation approach has been addressed by [80]. In that work an ISS, e.g., QEmu [81], executes the application and the operating

system, some HW components are mapped on the corresponding host devices while others are modeled in SystemC [45]. The communication between drivers and the corresponding devices modeled in SystemC is implemented through dedicated inter-process channels (i.e., sockets) leading to two main drawbacks: 1) HW/SW communication in case of SystemC-simulated devices is different from the final actual implementation since the designer has to put explicit socket calls in the driver implementation and in the SystemC device description; and 2) in case of multiple SystemC devices the number of sockets between QEmu and SystemC may decrease simulation speed.

The proposed co-simulation methodology described in this Section aims at solving these issues by supporting HW/SW communication directly in the ISS and in the HDL simulator. The advantages are: 1) the way in which device drivers access HW devices is the same both in case of host-mapped components and HDL models; and 2) a single inter-process channel is established between the ISS and the HDL simulator thus increasing the efficiency and scalability of the co-simulation framework which can handle several CPUs connected to many HDL models.

Co-simulation Architecture

Co-simulation is a methodology for the accurate verification of mixed HW/SW systems. It allows to meet the requirements for fast HW prototyping and for early SW development, because high level HW models can be effectively inserted into the development flow. The framework described in this work uses SystemC to model the HW and QEmu to emulate the SW even if the methodology can be applied to other similar tools. The reasons of the choice are: 1) QEmu already supports the use of host-mapped devices 2) SystemC supports the HW description at many abstraction levels, and 3) QEmu source code is available and easy to understand and modify.

QEmu: SW simulation

QEmu [81] achieves good SW simulation speed by using dynamic code translation to map SW instructions of the guest CPU to the host CPU so that it behaves as an ISS. QEmu has two operating modes:

- *Full system simulation.* In this mode, QEmu simulates a full system (for example a PC), including one or several processors and various peripherals; QEmu exploits the components present on the host platform to map the guest components.
- *User mode simulation.* In this mode, QEmu can launch processes compiled for one CPU on another CPU. It can be used to simplify cross-compilation and cross-debugging. Several processors are supported, e.g., x86, PowerPC, ARM, 32-bit MIPS, Sparc32/64 and ColdFire (i.e., m68k).

SystemC: HW simulation

HW devices are modeled in SystemC and module interfaces (i.e., input/output ports) follow the rules presented in [72]. The communication between the SystemC

simulator and the ISS is implemented by means of three new type of ports added to the SystemC library, i.e., `iss_in`, `iss_out`, `iss_inout` (in the following they are also grouped under the term `iss_port`) and `iss_interrupt`. The `iss_in` port is derived from the standard `sc_in` port and it is used to read data coming from ISS, the `iss_out` port extends the SystemC `sc_out` port and it allows to send data from SystemC to the ISS. These special ports can be used to model HW registers, thus allowing the ISS to read and write them. In the following text, we assume that HW registers are memory-mapped.

The connection between the two sides of the co-simulation is performed by binding specific addresses of the ISS memory space to SystemC `iss_port` contained in the HW models. When the CPU accesses some registers through their addresses, the SystemC kernel determines the corresponding special ports of the HW model. The link between SystemC ports and memory addresses in the ISS is implemented by using a binding table stored in the SystemC kernel.

Device driver structure

In actual embedded platforms, SW applications access HW devices through device drivers. The same mechanism is needed in a co-simulation model. According to good-practice rules [83] device drivers should follow a two-levels structure:

- The *II level device driver* contains the functions used by user applications to access the device (e.g., read, write, config, etc.). Each function implements a specific communication protocol according to the type of device.
- The *I level device driver* implements atomic operations used to access the device registers (such as `read` and `write`). These operations are invoked by the second level functions: the sequence of invocations forms the communication protocol. This level is equal for all devices except for some architectural choices (e.g., the addresses on which the device is mapped, interrupt handling, etc.).

Figure 4.16 shows an example of HW device and the corresponding driver code organized in two levels.

Co-simulation requirements

Since co-simulation is used for verification, the device drivers used in the co-simulation platform must be the same as on the actual operating system. This fact creates some requirements that must be met by the co-simulation architecture.

- The *CPU - device communication* mechanisms must be managed. The way used to access I/O depends on the computer architecture, bus and devices being used. However, the main mechanism used to communicate with devices is through memory-mapped I/O (MMIO) according to which specific areas of CPU's addressable space are reserved for I/O. Each I/O device responds to the CPU's access of device-assigned address space.
In the co-simulation architecture, access to MMIO regions must be managed by an external wrapper. Whenever the device driver accesses MMIO regions

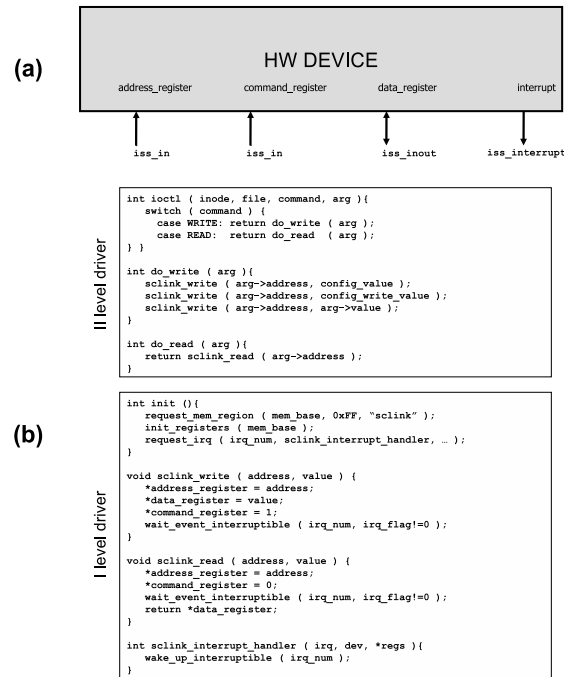


Fig. 4.16. HW device (a) and the corresponding driver code (b).

with read or write operations, the request must be forwarded to the simulated device and then the result must be brought back to the driver.

- Device drivers handle *interrupts*: thus, co-simulation must guarantee that interrupts risen by the SystemC device are forwarded to the ISS side of co-simulation.
- A device driver might contain *mutual exclusion* resources to avoid race conditions. Thus, co-simulation must manage concurrent access to the modules that handle communication with the SystemC side.

Co-simulation Methodology

Figure 4.17 shows how the SW application running on the QEmu simulator exchanges data with the SystemC simulator. An user application simply accesses the hardware devices by using their drivers, that read and write device registers through the I/O memory where the device is registered. Operations over this I/O memory pass through QEmu kernel virtualizing the hardware device implemented in SystemC. Communication with HW device, modeled in SystemC, is managed by SystemC kernel, suitably modified to support the co-simulation methodology. In order to implement this HW/SW simulator framework two steps are required:

- Modifications to the QEmu both to communicate with the SystemC simulator and to manage the HW device.
- Modifications to the SystemC simulator kernel. For the SystemC simulator, it is necessary to add the capability of reading and interpreting the messages

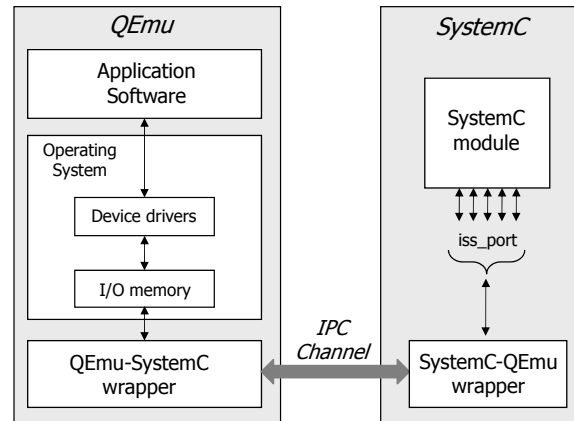


Fig. 4.17. QEmu-SystemC co-simulation schema.

coming from the QEmu side, as well as of sending interrupts to QEmu whenever the HW models generate them. These operations must be transparent to the designer who just writes the model by using the standard SystemC statements.

Communication between QEmu and SystemC simulator kernel is established by an inter-process channel (i.e., a socket) implementing the HW/SW interface in order to transmit synchronization messages.

SystemC-QEmu wrapper

The most meaningful parts of the SystemC - QEmu wrapper code are reported in Figure 4.18.

The SystemC - QEmu wrapper handles the SystemC side of co-simulation: it activates execution on the SystemC modules by setting the `iss_port` to the values received from QEmu through the socket.

Messages from QEmu are directed to four ISS ports on the SystemC side. The methods of the SystemC-QEmu wrapper are sensitive to these ports: whenever a data is received on a port, the corresponding method is called in order to update the wrapper registers and eventually trigger other events on the SystemC platform.

The wrapper consists of four main functions:

- `Read_iss_data_register`: this method is sensitive to the ISS data port. Whenever a new data is written to this port, the new value is saved in the data register;
- `Read_iss_address_register`: this method is sensitive to the ISS address port. Whenever a new data is written to this port, the new value is saved in the address register;
- `Read_iss_command_register`: this method is sensitive to the ISS command port. If a new data is written to this port, it means that the QEmu side has finished transmitting data and thus the SystemC side has already received the updated values for both data and address. The `read_iss_command_register`

function updates the command register and it writes 0 to the ISS control port, to keep the QEmu side waiting. Then, the `read_iss_command_register` wakes up the entry method by notifying a `run_io_process` event: the entry function will process the QEmu request and write the result to the ISS data port.

- **Entry:** this function waits for a `run_io_process` event. Whenever such an event is notified, the function writes the values of data, address and command on an `ahb_transport` port to the SystemC platform. The SystemC AHB bus will receive the data and forward it to the corresponding device. Then, the `entry` function gets the execution result from the `ahb_transport` port and it writes it to the ISS data port. Finally, the ISS control port is updated to 1, to notify QEmu that execution on SystemC side is finished. This value will raise an interrupt on the QEmu side.

```

void read_iss_command_register () {
    iss_command_register->read ( tmp, sizeof (int) );
    command_register = tmp;
    iss_command_register->write ( 0, sizeof (int) );
    run_io_process.notify();
}
void read_iss_address_register () {
    iss_address_register->read ( tmp, sizeof (int) );
    address_register = tmp;
}
void read_iss_data_register () {
    iss_data_register->read ( tmp, sizeof (int) );
    data_register = tmp;
}
void entry () {
    wait (run_io_process);
    switch (command_register) {
        case WRITE: request = set_request ( address_register,
            data_register, WRITE );
            response = ahb_transport ( request );
        case READ: request = set_request ( address_register,
            data_register, READ );
            response = ahb_transport ( request );
            iss_data_register->write
                ( response->data,sizeof(int) );
    }
    iss_control_register->write ( 1 ,sizeof (int) );
}

```

Fig. 4.18. SystemC - QEmu wrapper code.

QEmu-SystemC wrapper

The most meaningful parts of the Qemu - SystemC wrapper code are reported in Figure 4.19. The Qemu - SystemC wrapper has an important role in the co-simulation architecture: it manages accesses to MMIO regions assigned to devices, forwarding the requests to the SystemC side and bringing the result back to the device driver.

The wrapper consists of six main functions:

- **Update:** this function is used to raise an interrupt or to knock an interrupt down;
- **Init:** this function is used to initialize memory and I/O resources, to manage the addresses of the ISS ports on the SystemC side and to start the socket communication;

- **Restore**: it restores the socket communication;
- **Read**: this function is invoked whenever a read operation is performed by the driver on the I/O memory assigned to the device. This function prepares the data to be sent via socket to the SystemC side and it invokes the `cosim` function. Then, it returns the result received via socket;
- **Write**: this function is invoked whenever a write operation is performed by the driver on the I/O memory assigned to the device. This function prepares the data to be sent via socket to the SystemC side and it invokes the `cosim` function;
- **Cosim**: this function sends data to the SystemC side via socket. It is invoked by both the `read` and the `write` functions.

```

void update ( *state ){
    set_irq_new( pic, irq, state->flag != 0 );
}

int read ( *state, addr ){
    switch ( addr ) {
        case 0 : cosim ( 0, DATA_PORT, 0 );
                return state->data;
        case 1 : cosim ( 0, ADDR_PORT, 0 );
                return state->addr;
        case 2 : cosim ( 0, CONTROL_PORT, 0 );
        case 3 : state->irq_pending = 1;
                cosim ( 0, COMM_PORT, 0 );
    } }

void write ( *state, addr, val ){
    switch ( addr ) {
        case 0 : cosim ( 1, DATA_PORT, val );
        case 1 : cosim ( 1, ADDR_PORT, val );
        case 2 : cosim ( 1, CONTROL_PORT, val );
        case 3 : cosim ( 1, COMM_PORT, val );
                while cosim ( ( 0, CONTROL_PORT, 0 ) != 1 );
                state->flag = 1;
                update( *state );
    } }

int cosim ( op, addr, data ) {
    msg = msg_create ( op, addr, data );
    send ( socket, msg, sizeof ( msg ), 0 );
    rcv ( socket, msg, sizeof ( msg ), 0 );
}

```

Fig. 4.19. QEmu - SystemC wrapper code.

Execution flow

The execution flow to access a simulated device is the following:

1. A user application wants to access the device. Thus, it invokes the `ioctl` function of the corresponding second level device driver.
2. The second level device driver implements the communication protocol with a certain number of invocations of the functions implemented by the first level device driver (`sclink.write` and `sclink.read`).
3. The first level device driver writes data to the MMIO locations assigned to the device. Then, it invokes the `wait_event_interruptible` function to suspend its execution until an interrupt is received.

4. QEmu catches the accesses to the MMIO locations and invokes the functions of the QEmu-SystemC wrapper to forward the requests to the SystemC side of co-simulation. When the SystemC-QEmu wrapper receives a command value, the simulated device is activated.
5. As soon as the requested operation has been executed, the SystemC-QEmu wrapper sends an acknowledge message to the QEmu side of co-simulation. This message is interpreted as an interrupt: thus, the QEmu-SystemC wrapper functions notifies that execution is finished by rising an interrupt.
6. The interrupt is forwarded to the target CPU.
7. The target CPU invokes the interrupt handler function (`sclink_interrupt_handler`), that knocks the interrupt down and executes the `wake_up_interruptible` function.
8. The first level device driver resumes execution and it returns the result to the second level device driver.
9. As soon as the second level device driver has completed the communication protocol, it returns the final result to the user application, that resumes execution.

Figure 4.20 shows and clarifies the main steps of the execution flow.

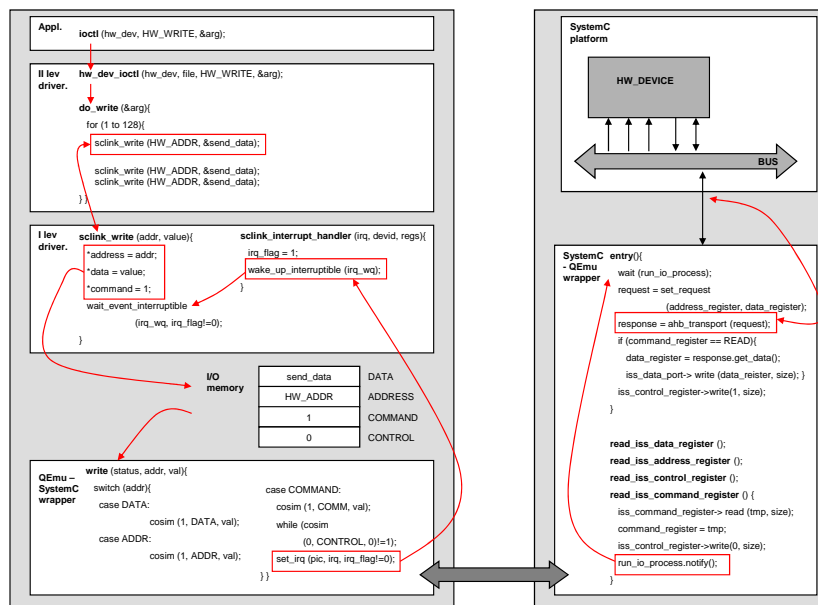


Fig. 4.20. Execution flow.

Model-level	Tool-level	Speed up
2520 sec	120 sec	20X

Table 4.1. Co-simulation performance.

Experimental analysis

The proposed HW/SW (SystemC/QEmu) co-simulation methodology has been evaluated in two different experiments. The former compares the tool-level approach with respect to the model-level approach. The latter shows the validity of the methodology to model an actual platform.

Co-simulation performance

The first experiment regards co-simulation of a video MPEG-2 decoder. Software decoder has been executed by QEmu and uses HW SystemC modules to compute Inverse Discrete Cosine Transform (IDCT) function. The co-simulation methodology described in this paper allows to rely on SystemC kernel for the communication between QEmu and SystemC. Using this approach the performance is increased of about 20 times, as shown in Table 5.1, compared with the approach where communication issues are managed at SystemC model level as described in [80].

Co-simulation of modeled and actual devices

The second experiment concerns the model of a ZigBee/802.15.4 wireless scenario taken from the Angel European project [82]. As depicted in Figure 4.21.a, Node 0 and Node 1 transmit information to Node 2 through a wireless channel. Figure 4.21.b shows the partitioning of the scenario onto the simulation tools. Node 1 and Node 2 are modeled in SystemC at TLM level; also the wireless channel is reproduced in SystemC by using the SCNSL library [73]. SW running on Node 0 is simulated by QEmu while HW devices are mapped both in SystemC and on the host components. At SystemC side a wireless network interface is implemented by a TLM module of the serial interface (UART) and of the RF-module. An application software transmits packets to the network by sending appropriate commands through the serial interface accessed by the corresponding device driver.

The user application is the same in both cases. It initializes the wireless network and then it sends a packet every 3 seconds. Since the application has a regular time behavior, the validation of the simulated model against the actual platform can be done by comparing the execution time for both cases as a function of the number of packets sent. This comparison is reported in Table 4.2 which shows a negligible increase of the execution time (about 0.6%) in case of co-simulation.

4.2 AME_2 design level

At AME_2 level a System/Network partitioning is applied to the model designed during the AME_3 level; modules are mapped onto network nodes and communications between nodes are provided by AME_2 services through a network simulator.

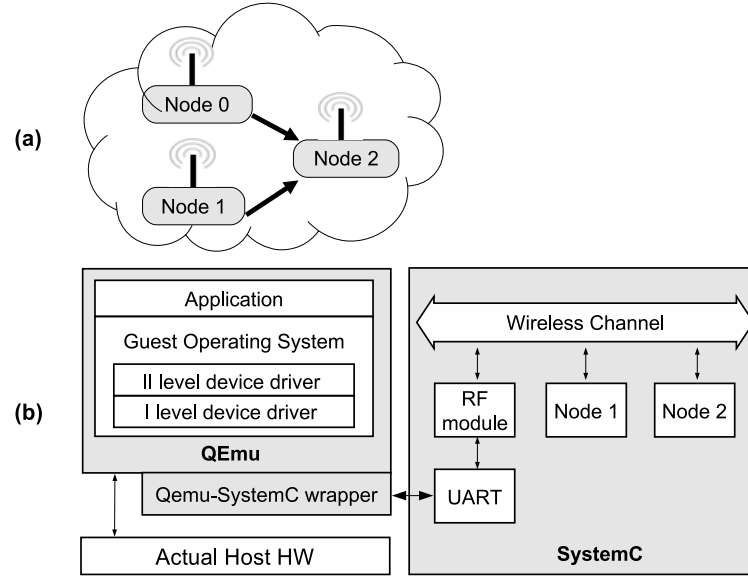


Fig. 4.21. Co-simulation of a real scenario.

Traffic load	Actual platform	Co-simulation model
10 packets	44.101 sec	44.517 sec
20 packets	75.125 sec	75.672 sec
40 packets	137.062 sec	137.879 sec

Table 4.2. Comparison of the execution time of the actual platform and its co-simulation model.

Therefore a simulated network infrastructure is involved in the whole framework at AME_2, as shown in Figure 1.2.b. AME_2 provides the same API of the previous design step, even if opportunely modified to establish a communication with a network simulator (e.g., NS2, SCNSL [73]). Exploiting the same API, the NES designer can simulate the same applications modelled during the AME_2 design step, but evaluating the NES applications impact on the network requirements. The simulation environment is connected to the AME_2 by using a general communication interface named Network Simulator Interface (NSI). This feature guarantees to use different network simulators. Doing that, it's possible to involve the network simulator providing the protocol required. Figure 4.22 shows the AME_2 architecture involving the NSI to establish the communication with the simulator.

4.2.1 Network Simulator Interface

The network simulator interface (NSI) serves as an abstraction layer hiding the different NES implementations peculiarities from end-user applications. NSI implements two main tasks:

- accomplish the application functionalities sending/receiving data to the right network nodes.

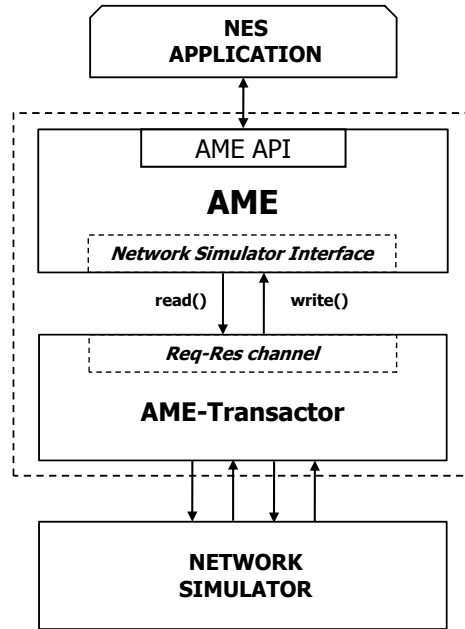


Fig. 4.22. Interaction between AME and Network Simulator.

- establish a communication with a network simulator

The first point is implemented by using a XML packet, a standardized and flexible well-formed meta-language, capable of describing data information in a structured and portable manner. The packet, named `MWpkt` includes the following tags:

- `MWfrom` represents the data source point.
- `MWto` indicates the data destination point.
- `op_type` describes the operation type involved in the packet (e.g., Tuplespace `read` operation, MOM `publish` operation, etc.).
- `data` contains the information specific related to the operation type.

4.2.2 AME-Transactor

The communication between AME and the network simulator is implemented by using the communication interfaces proposed by OSCI (Open SystemC Initiative) TLM (Transaction-Level Modeling) standard. The intention of OSCI TLM is to coordinate different teams of engineers in the modelling of the same design through an interoperable interface. Because of this teams divide, it is often useful to define a protocol specific boundary between these groups of engineers. This interface is sometimes called the convenience interface. It will typically consist of methods that make sense to users of the protocol in question (e.g., `read/write`). A user will use initiator ports that supply these interfaces, and define target modules which inherit

from these interfaces. The infrastructure team will implement the protocol layer for the users. This consists of the request and response classes that encapsulate the protocol. Therefore, the use of OSCI TLM interface guarantees standard, efficient and safe exchange of transactions between AME and the network simulator.

The AME-Transactor interface is modelled by using the following TLM channel.

$$tlm_req_rsp_channel < REQ, RSP >$$

The `tlm_req_rsp_channel<REQ,RSP>` class consists of two fifos, one for the request going from initiator (AME) to target (AME-Transactor) and the other for the response being moved from target (AME-Transactor) to initiator (AME). To provide direct access to these fifos, it exports the put request and get response interfaces to the initiator (as shown in Figure 4.22) and the get request and put response interfaces to the target. This channel adopts a standard TLM packet involving a generic payload for the data exchanged between initiator and target. AME loads the XML packet previously described inside the TLM packet in order to send information to the network simulator.

AME-Transactor is an intermediate layer that acts as transactor and data conversion layer. The transactor is a translator from different hardware abstraction level and permits to connect AME2, implemented at TL3, with a Network simulator implemented at different abstraction level (e.g. SCNSL_TLM or SCNSL_RTL). Data conversion is also required to convert AME packet into the specific packet managed from network simulator.

It's important to understand that AME-Transactor doesn't depend from middleware programming paradigm used by the application, but from Network simulator. A NES designer can connect the preferred Network Simulator to AME just modelling the related AME-Transactor able to establish the correct data conversion with the Network Simulator.

4.2.3 SCNSL-AME-Transactor

This Section reports the AME-transactor implementation used to connect SystemC Network Simulation Library (SCNSL) to AME.

SCNSL is a free simulation kernel of Networked Embedded Systems, written in SystemC and C++. This library allows to model network scenarios in which different kinds of nodes, or nodes described at different abstraction levels, interact together. The use of SystemC as unique tool has the advantage that HW, SW, and network can be jointly designed, validated and refined.

To support network modeling and simulation, SCNSL provides the following elements:

- Kernel: the kernel is responsible for the correct simulation, i.e., its adherence to the behavior of an actual communication channel; the kernel executes events in the correct temporal order and it has to take into account the physical features of the channel such as, for example, propagation delay, signal loss and so forth;
- Node: nodes are the active elements of the network; they produce, transform and consume transmitted data;
- Packet: in packet-switched networks the packet is the unit of data exchanged among nodes; it consists of a header and a payload.

- Channel: the channel is an abstraction of the transmitting medium which connects two or more nodes; it can be either a point-to-point link or a shared medium.
- Port: nodes use ports to send and receive packets.

More details can be found in [73].

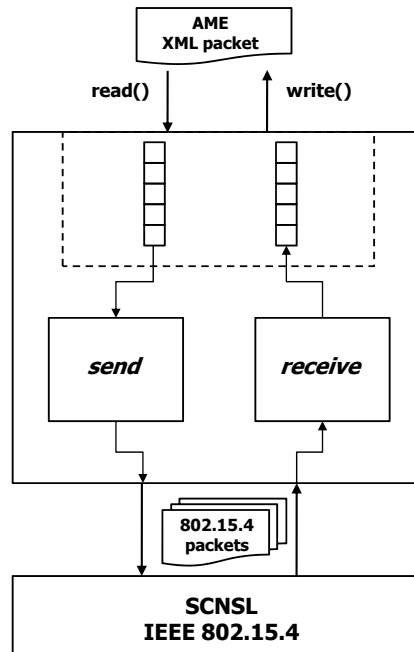


Fig. 4.23. SCNSL-AME-Transactor implementation.

SCNSL-AME-Transactor, shown in Figure 4.23, consists of two main functions for the transmission from AME to SCNSL and viceversa, called *send* and *receive* respectively. *Send* function retrieves data packets coming from AME and splits them into a sequence of messages depending on the maximum transmission unit (MTU) supported by the network protocol simulated by SCNSL. MTU value for SCNSL 802.15.4 model is equal to 124 byte. On the other hand, *Receive* function waits messages from SCNSL and rebuilds the AME XML packet containing the programming paradigm features used by the designer. Finally this packet is sent to AME.

4.2.4 NS2-AME-Transactor

NS-2 is a discrete event simulator targeted at networking research. It covers a very large number of applications, protocols, network types, network elements and traffic models. NS-2 provides substantial support for simulation of TCP, routing, and

multicast protocols over wired and wireless (local and satellite) networks. NS-2 simulator is based on two languages: an object oriented kernel simulation, written in C++, and an OTcl (an object oriented extension of Tcl) interpreter, used to execute user's command scripts. More details can be found in [53].

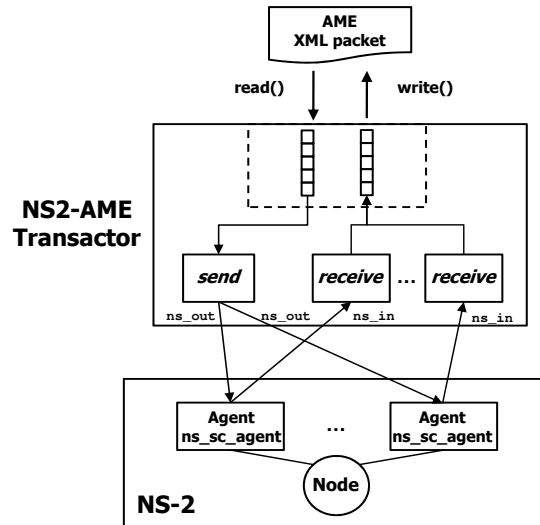


Fig. 4.24. NS2-AME-Transactor implementation.

NS2-AME-Transactor allows to involve NS-2 simulator in AME environment in order to simulate network aspects. The connection between NS-2 and AME exploits the co-simulation methodology describes in Section 4.1.4. The co-simulation is implemented at *Agent* level.

Figure 4.24 depicts the NS2-AME-Transactor. It receives AME XML packet coming from AME NSI layer; these packets are elaborated by the *Send* function in order to send them to the appropriate *Agent/ns_sc_agent* modelled in NS-2. No payload split into a sequence of messages is needed to implement (as described in Section 4.2.3 concerning the SCNSL-AME-Transactor) because this operation is performed by the NS-2 simulator depending on the network protocol simulated (TCP, UDP, 802.11, etc.). The connection with NS-2 is established by using *ns_in* port.

On the other hand, a *Receive* function for each NS-2 *Agent/ns_sc_agent* is created. The *Receive* waits messages from NS-2 and delivers them to AME able to send the messages to the application. The connection with NS-2 is established by using *ns_out* port.

It is worth noting that on the NS-2 side the Tcl network topology has to be modeled by using *Agent/ns_sc_agent*; in fact, this Agent allows the co-simulation with SystemC as described in Section 4.1.4.

4.3 AME_1

At AME_1 design level *HW/SW partitioning* is applied to each node to map functionalities to HW and SW components accordingly to several constraints (e.g., performance, cost, and component availability). HW and SW components are simulated by SystemC and interact with the network model as in the previous stage. Also the software is simulated by SystemC. AME_1 API services are the same of the previous design steps, but the implementation is different to communicate with the network simulator, simulating the *Network model*, as shown in Figure 1.2 and in Figure 3.4.

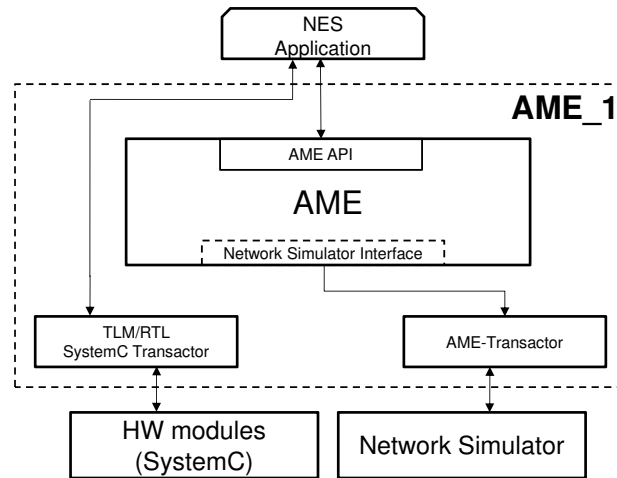


Fig. 4.25. TLM/RTL transactor to include HW models.

Figure 4.25 shows the transactor to involve *Hardware model* inside the AME environment. The transactor is needed in order to link modules, IPs, designed at different levels of abstraction (Transaction Level Modeling or Register Transfer Level). Transaction Level Modeling (TLM) modules communicate with each other through function calls and allow the designers to focus on the functionality, while abstracting away implementation details. At the Register Transfer Level (RTL) different modules communicate through pin level signaling. SoC design methodologies involve the integration of different intellectual property (IP) blocks modeled at different levels of abstraction. TLM/RTL transactor has to be modeled using a finite state machine (FSM) providing a functional specification of the protocol's behavior

Figure 4.25 emphasizes that the application software changes with respect to the AME.2 design level. Obviously, the *HW/SW partitioning* applied in this phase imposes the application software has to establish a communication with the hardware modules generated. For instance, let's suppose to simulate a WSN node

running an application software sensing data from an accelerometer sensor and delivering the data just obtained to another remote WSN node.

At AME_3 level, where neither network nor hardware aspects are involved in the simulation, the application software can be modelled simulating the sensing accelerometer data by using a random process.

At AME_2 level, the network is involved in the simulation to model the wireless protocol (e.g., IEEE 802.15.4). The communication between AME_2 and the simulated wireless protocol is implemented by using the AME-Transactor described in Section 4.2.2. In this case, the application software remains un-changed because the *System/Network partitioning* does not concern the application software. Therefore, also in this design phase, the application software can be modelled simulating the sensing accelerometer data by using a random process.

Finally, at AME_1 level, due to the *HW/SW partitioning* the application software has to be changed in order to communicate with a HW module simulating the accelerometer sensor modelled in SystemC language; the communication between the application software and the accelerometer sensor is implemented by using the TLM/RTL transactor.

4.4 Actual or Simulated platform

At the conclusion of the *Refinement* process, when the AME-centric design flow is completed, a mapping process is implemented to deploy the simulated application over the actual or simulated platform as shown in Figure 4.26. If the HW/SW resources of the actual/simulated platform allow the presence of an actual middleware, the the application code modelled by using AME is automatically translated to replace calls to AME services with calls to the actual middleware. This procedure called *Mapping* process is described in Section 6.

The application designer can deploy the actual application directly on the actual platform or can exploit the HW/SW/Network environment, described in Section 4.1, to execute the actual application on the simulated platform.

The HW and Network aspects of the simulated platform can be modelled by using the co-simulation environment described in Section 4.1.4.

Concerning the SW layer, it includes the actual application, the actual middleware and an operating system; the latter is not mandatory and typically it depends on the NES. The SW layer can be run by using the co-simulation environment described in Section 4.1.5 and in Section 4.1.6.

4.5 Experimental analysis

An example of NES-based application is shown in Figure 4.27 which represents a remotely-assisted training session; expert trainers and physicians can monitor a user through a NES acting as gateway between a short-range wireless network (e.g., ZigBee or Bluetooth) and Internet. The user wears a 3D acceleration sensor which interacts with the gateway to compute the step rate and the run speed.

Step rate can be obtained from acceleration data at the cost of some computations [59]; however, such conversion reduce the amount of data to be transmitted

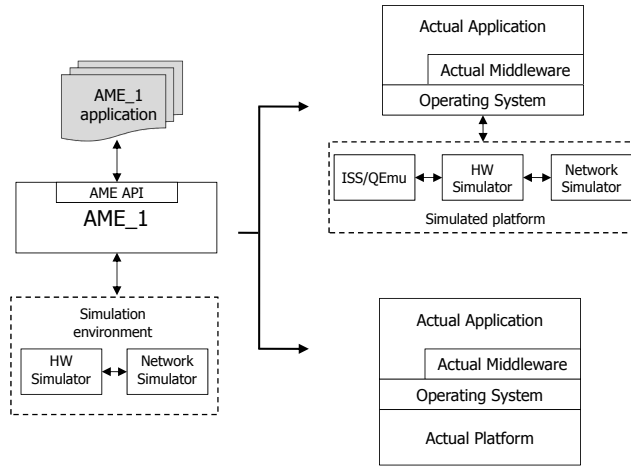


Fig. 4.26. Mapping process onto the actual or the simulated platform.

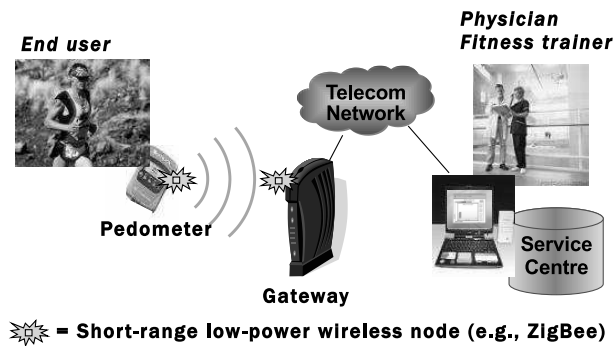


Fig. 4.27. NES-based case study.

since a sequence of 3D acceleration values are reduced to a single step rate value. Some general observations about NES-based applications can be extracted from this simple example:

- several NES's cooperate through the network to accomplish a single task;
- the task can be decomposed into smaller interacting subtasks which can be assigned to different NES's (aka network nodes);
- the assignment of subtasks to network nodes affects the required resources on the nodes and the demand of bandwidth on the communication channels among them;
- transmission bitrate and computational power affect energy consumption which may be critical in case of battery-powered devices;
- the computational requirements for each NES affect their cost.

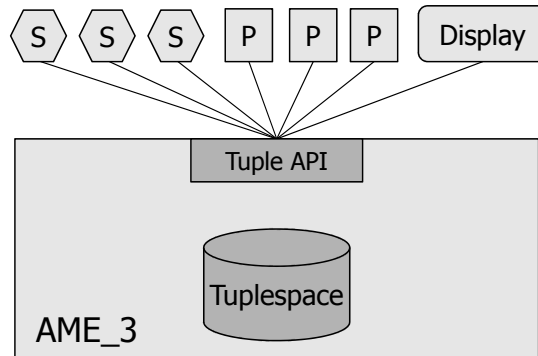


Fig. 4.28. AME.3 model of the step-counter application with three user.

The *Refinement* process proposed in this Section aims at providing a methodology to address these aspects in the design of NES-based applications. It has been applied to the example of NES-based application shown in Figure 4.27. Figure 4.28 shows the model at AME_3 with three users. For each user a sensor module (labelled with "S" representing the accelerometer sensor) and a processing module for step detection (labelled with "P" implementing the pedometer module) are instantiated while a `Display` module simply shows training results for all the users. Modules communicate through primitives provided by the Abstract Middleware which reproduces the behavior of a *Tuplespace* programming paradigm. The step detection algorithm has been taken from an application note by Analog Devices [59]; the algorithm was specified in C language and thus the introduction in the SystemC model has been straightforward.

This scenario has been implemented in AME_3 as reported in the pseudo-code of Figure 4.29. Tuplespace programming paradigm has been used and the specific API (`read`, `write`, `take`) have been highlighted in bold face style. Figure 4.29.4 represents the application code running on the "S" module (`accelerometer`). It transmits the X/Y/Z information to the related "P" module (`pedometer`) by writing a tuple containing the X/Y/Z data; this operation is implemented by using the `write` service.

Figure 4.29.3 represents the application code running on the "P" module (`pedometer`) which extracts (`take` service) available X/Y/Z information written by the associated "S" module and executes the step detection algorithm (`StepCounter`) to calculates the "step" and the "distance" value. Moreover, it write these values (`< STEP, DISTANCE >`) in the tuplespace through the `write` service.

Figure 4.29.2 represents the application code running on the `Display` module. It extracts the `< STEP, DISTANCE >` tuple and prints it.

Finally, Figure 4.29.1 describes the instantiation of the all actors. It shows the instantiation of the three "S" modules (`a.1`, `a.2`, `a.3`), the instantiation of the three related "P" modules (`p.1`, `p.2`, `p.3`), the instantiation of the `Display`

<pre> int sc_main(int argc, char *argv[]){ Acc *a_1 = new Acc("Acc1"); Acc *a_2 = new Acc("Acc2"); Acc *a_3 = new Acc("Acc3"); Pedometer *p_1 = new Pedometer ("Ped1"); Pedometer *p_2 = new Pedometer ("Ped2"); Pedometer *p_3 = new Pedometer ("Ped3"); Display *d = new Display("Display"); AME_3 *mw=new AME_3("AME_3"); a_1->mw_port(mw->tuple_port); a_2->mw_port(mw->tuple_port); a_3->mw_port(mw->tuple_port); p_1->mw_port(mw->tuple_port); p_1->setAccelerometer(a_1->name); p_2->mw_port(mw->tuple_port); p_2->setAccelerometer(a_2->name); p_3->mw_port(mw->tuple_port); p_3->setAccelerometer(a_3->name); display->mw_port(mw->tuple_port); sc_start(-1); return 0; }; </pre>	<pre> SC_MODULE(Pedometer) { /* pedometer.h */ sc_port<mw_tuple_if> mw_port; void run(); void setAccelerometer(string acc_name); SC_CTOR(Pedometer) { SC_THREAD(run); }; void Pedometer::run() { /* pedometer.cc */ while (1) { mw_port->take(<X, Y, Z>); StepCounter (<X, Y, Z>); if (IsNewStep()) mw_port->write(<<STEP, DISTANCE>>) } }; }; SC_MODULE(Acc) { /* acc.h */ sc_port<mw_tuple_if> mw_port; void run(); SC_CTOR(Acc) { SC_THREAD(run); }; void Acc::run() { /* acc.cc */ int X = DATA X; int Y = DATA Y; int Z = DATA Z; mw_port->write(<<X, Y, Z>>) }; }; </pre>
<pre> SC_MODULE(Display) { /* display.h */ sc_port<mw_tuple_if> mw_port; void run(); SC_CTOR(Display) { SC_THREAD(run); }; }; void Pedometer::run() { /* display.cc */ while (1) { mw_port->take(<<STEP, DISTANCE>>) printf(STEP, DISTANCE); } }; </pre>	

Fig. 4.29. AME_3 model pseudo-code of the step-counter application.

module and finally the creation of the AME environment (mw). Each module is connected to the AME environment through the tuple_port.

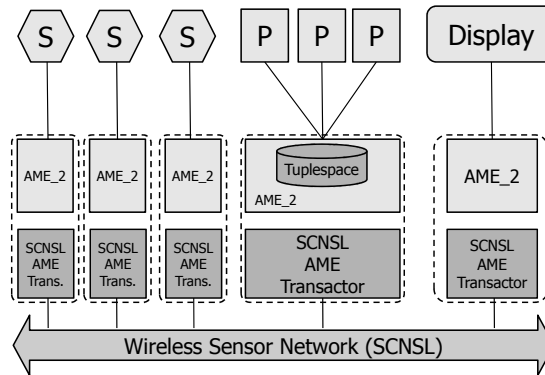


Fig. 4.30. AME_2 model of the step-counter application with three users (case 1).

Figure 4.30 and Figure 4.31 show two different ways to refine the model from AME_3 to AME_2. Dashed boxes represent distinct network nodes interacting together through a subset of the well-known IEEE 802.15.4 wireless standard, i.e., peer un-slotted transmissions with acknowledge. In Figure 4.30, each user wears a wireless accelerometer while step detection is performed in the gateway.

In Figure 4.31, all the accelerometers are connected to the same node while step detection is performed in dedicated nodes. In both cases, a dedicated node hosts the display subtask.

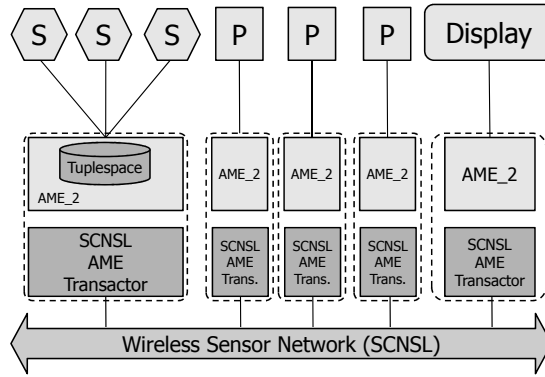


Fig. 4.31. AME_2 model of the step-counter application with three users (case 2).

Figure 4.32 reports the implementation of the step-counter application with three users (case 1) simulated by using AME_2 environment. In this scenario, SCNSL is involved in order to simulate the whole NES application on the IEEE 802.15.4 wireless protocol.

Figure 4.32 just shows the SystemC pseudo-code to model the scenario; "S", "P" and Display modules remain un-changed with respect to the AME_3 implementation. Lines 9-11 emphasize the instantiation of one AME_2 module for each "S" module application (wireless accelerometer). The instantiation of AME_2 module for the Display module is shown in Line 12. Line 13 reports the creation of the AME_2 module to connect the gateway simulating the step detection (pedometers) algorithm. The connection between the modules is implemented in the pseudo-code at Line 15-24.

Lines 28-35 show the insertion of the SCNSL-AME-Transactor for each AME_2 module previously created; this module allows to connect SCNSL network simulator (Line 36) simulating IEEE 802.15.4 protocol (Line 39) to AME_2 (Lines 30, 31).

Figure 4.33 shows the modelling of the step-counter application where all the accelerometers are connected to the same node while step detection is performed in dedicated nodes (case 2 depicted in Figure 4.31). In this case all "S" module (a_1, a_2, a_3) are connected to the same AME_2 middleware (name mw_acc) as reported in Line 15, 16 and 17.

Each "P" module (p_1, p_2, p_3) implementing the step detection is connected to a different AME_2 middleware in order to simulate a dedicated node (Lines 18-23).

```

1 int sc_main(int argc, char *argv[]){
2 Acc *a_1 = new Acc("Acc1");
3 Acc *a_2 = new Acc("Acc2");
4 Acc *a_3 = new Acc("Acc3");
5 Pedometer *p_1 = new Pedometer ("Ped1");
6 Pedometer *p_2 = new Pedometer ("Ped2");
7 Pedometer *p_3 = new Pedometer ("Ped3");
8 Display *d = new Display("Display");
9 AME_2 *mw_acc1=new AME_2("AME_2");
10 AME_2 *mw_acc2=new AME_2("AME_2");
11 AME_2 *mw_acc3=new AME_2("AME_2");
12 AME_2 *mw_display=new AME_2("AME_2");
13 AME_2 *mw_pedometer=new AME_2("AME_2");
14 #define TUPLE_SERVER mw_pedometer

15 a_1->mw_port(mw_acc1->tuple_port);
16 a_2->mw_port(mw_acc2->tuple_port);
17 a_3->mw_port(mw_acc3->tuple_port);
18 p_1->mw_port(mw_pedometer->tuple_port);
19 p_1->setAccelerometer(a_1->name);
20 p_2->mw_port(mw_pedometer->tuple_port);
21 p_2->setAccelerometer(a_2->name);
22 p_3->mw_port(mw_pedometer->tuple_port);
23 p_3->setAccelerometer(a_3->name);

24 display->mw_port(mw_display->tuple_port);

25 foreach mwAME2 in [mw_acc1,mw_acc2, mw_acc3,mw_display,mw_pedometer]
26     mwAME2->setMwSpaceServer(TUPLE_SERVER)

27 sc_clock clock ("clock", sc_time(1,SC_MS));

28 ReqRsp *channel[TOT_MW*2]=new ReqRsp("channel")
29 foreach mwAME2 in [mw_acc1,mw_acc2,mw_acc3,mw_display,mw_pedometer]
30     mwAME2->net_out(req_rsp[t]->put_request_export)
31     mwAME2->net_in(req_rsp[t+1]->get_request_export)

32 SCNSL-AME_trans *transactor[TOT_MW]=new SCNSL-AME_trans("trans")
33 foreach transactor in transactor[TOT_MW]
34     transactor->mw_in(req_rsp[t]->get_request_export)
35     transactor->mw_out(req_rsp[t+1]->put_request_export)

36 SCNSL *network = new SCNSL("wnet")
37 network->setClock(clock)
38 foreach transactor in transactor[TOT_MW]
39     transactor->setMacNode(network->802_15_4mac)
40 ...

41 sc_start(-1);
42 return 0;
43 };

```

Fig. 4.32. AME_2 model pseudo-code of the step-counter application (case 1).

Finally, the SCNSL-AME-Transactor creation to involve the SCNSL simulating the IEEE 802.15.4 protocol is performed (Lines 28-35).

We simulated 20 s of operation of the distributed application as a function of the task-node assignment and the number of accelerometer sensors. Table 5.1 compares the performance of the two network configurations. We considered the total number of packets which are sent to the network, the average transmission delay of packets and the number of transmissions per packet (average and max

```

1  int sc_main(int argc, char *argv[]){
2  Acc *a_1 = new Acc("Acc1");
3  Acc *a_2 = new Acc("Acc2");
4  Acc *a_3 = new Acc("Acc3");
5  Pedometer *p_1 = new Pedometer ("Ped1");
6  Pedometer *p_2 = new Pedometer ("Ped2");
7  Pedometer *p_3 = new Pedometer ("Ped3");
8  Display *d = new Display("Display");
9  AME_2 *mw_acc=new AME_2("AME_2");
10 AME_2 *mw_ped1=new AME_2("AME_2");
11 AME_2 *mw_ped2=new AME_2("AME_2");
12 AME_2 *mw_ped3=new AME_2("AME_2");
13 AME_2 *mw_display=new AME_2("AME_2");
14 #define TUPLE_SERVER mw_acc

15 a_1->mw_port(mw_acc->tuple_port);
16 a_2->mw_port(mw_acc->tuple_port);
17 a_3->mw_port(mw_acc->tuple_port);
18 p_1->mw_port(mw_ped1->tuple_port);
19 p_1->setAccelerometer(a_1->name);
20 p_2->mw_port(mw_ped2->tuple_port);
21 p_2->setAccelerometer(a_2->name);
22 p_3->mw_port(mw_ped3->tuple_port);
23 p_3->setAccelerometer(a_3->name);

24 display->mw_port(mw_display->tuple_port);

25 foreach mwAME2 in [mw_acc,mw_ped1, mw_ped2,mw_ped3,mw_display]
26     mwAME2->setMwspaceServer(TUPLE_SERVER)

27 sc_clock clock ("clock", sc_time(1,SC_MS));

28 ReqRsp *channel[TOT_MW*2]=new ReqRsp("channel")
29 foreach mwAME2 in [mw_acc,mw_ped1,mw_ped2,mw_ped3,mw_display]
30     mwAME2->net_out(req_rsp[t]->put_request_export)
31     mwAME2->net_in(req_rsp[t+1]->get_request_export)

32 SCNSL-AME_trans *transactor[TOT_MW]=new SCNSL-AME_transactor("trans")
33 foreach transactor in transactor[TOT_MW]
34     transactor->mw_in(req_rsp[t]->get_request_export)
35     transactor->mw_out(req_rsp[t+1]->put_request_export)

36 SCNSL *network = new SCNSL("wnet")
37 network->setClock(clock)
38 foreach transactor in transactor[TOT_MW]
39     transactor->setMacNode(network->802_15_4mac)
40 ...

41 sc_start(-1);
42 return 0;
43 };

```

Fig. 4.33. AME.2 model pseudo-code of the step-counter application (case 2).

value, respectively). Case 1 exhibits a higher delay but a lower number of retransmissions with respect to Case 2.

Figures 4.34, 4.35, 4.36 show the described metrics as a function of the number of acceleration sensors. Case 1 provides a more scalable solution than Case 2; in fact the former allows up to ten sensors while the latter saturates the channel capacity with six sensors.

	Sent packets	Avg. packet delay (s)	Transmissions/packet	
			Avg.	Max.
Case 1	2303	0.008	2.09	14
Case 2	79	0.004	11.91	565

Table 4.3. Network performance with six users as a function of the task-node assignment.

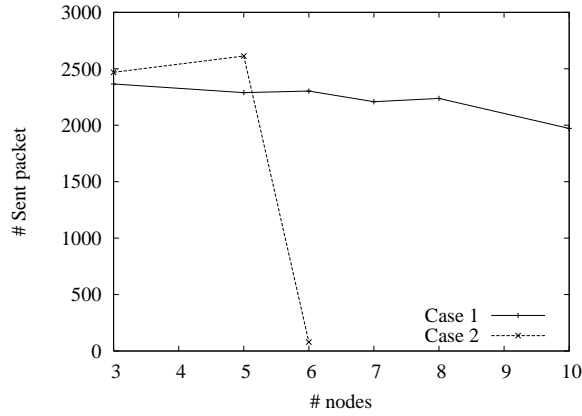


Fig. 4.34. Network performance as a function of the task-node assignment and of the number of acceleration sensors: Sent packets.

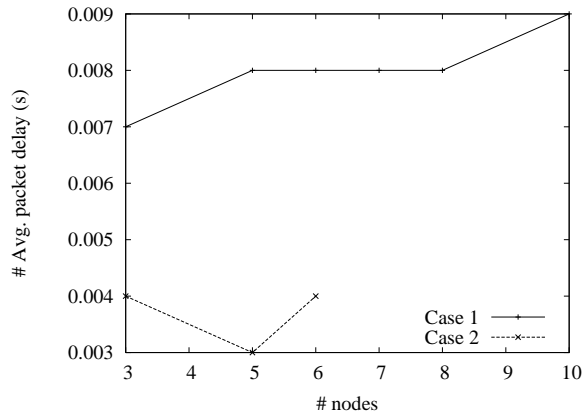


Fig. 4.35. Network performance as a function of the task-node assignment and of the number of acceleration sensors: Avg. packet delay.

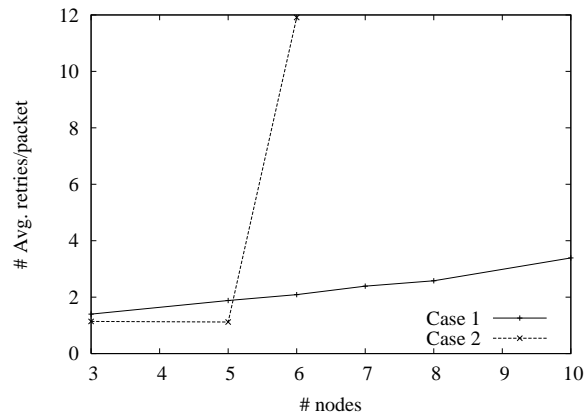


Fig. 4.36. Network performance as a function of the task-node assignment and of the number of acceleration sensors: Avg. number of transmissions/packet.

Translation

In this chapter the translation process between the different programming paradigms provided by AME is described. A proxy-based mechanisms (called AME Proxy Middleware) is implemented to allow the designer to smoothly move across different programming paradigms in order to validate and simulate the NES applications. Finally, the proposed translation mechanism has been applied to a house temperature monitor scenario to evaluate the effectiveness of the proposed solution, pointing up the advantages of the proxy-based middleware solution.

5.1 AME Proxy Middleware

A new module is automatically generated to implement the translation between the different programming paradigms provided by AME (e.g., MOM to Tuplespace and viceversa). This module is called proxy-MW module, as depicted in Figure 5.1 and it will become a real part of the final application. The proxy-MW module provides the same Tuplespace API provided by AME and it represents a proxy layer between the applications and AME.

Let us suppose to translate a MOM application into a Tuplespace application (the translation methodology will be described in Section 5.7). The proxy-MW module interacts with the application above using a Tuple-space paradigm and it translates every operation into MOM services. On the middleware side, the proxy-MW module still uses the MOM interface. In this way, applications think to dialogue with the middleware by using a tuple-space interface, but in truth they dialogue with proxy-MW. Thus, AME provides a X-paradigm interface, using the services of a Y-paradigm. Data is stored inside of the middleware in the Y-paradigm format. In this way, Y-paradigm applications will access data it without noticing that the application writing them follows a different paradigm.

The transparency enables to design an AME application which is composed by several parts, each one written by using different program-paradigms. Moreover, when the application accesses to the data, consistence will be always assured. Another advantage of this approach is that the automatically program-paradigms translation does not need of a SystemC parser, easing the translation mechanism. This feature also guarantees a reduction of the translated applications in term of

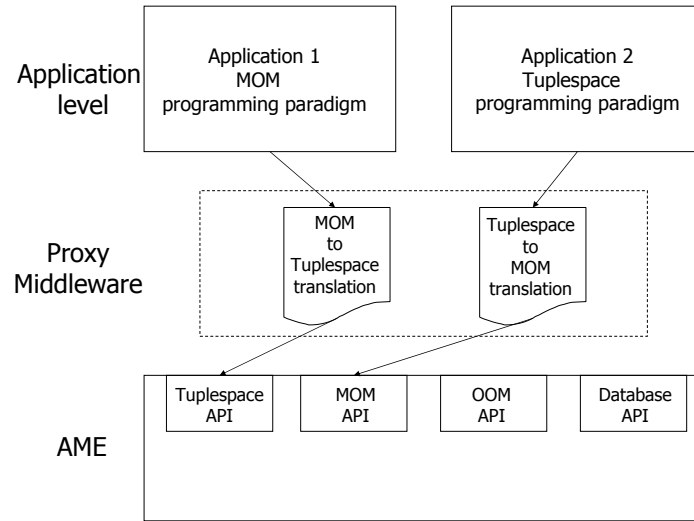


Fig. 5.1. AME Proxy Middleware.

code lines, because it isn't necessary to replace the translation-paradigm code in each point of the starting code.

Finally, proxy-MW makes easier the mapping process since the programming paradigm must be not the same in the transfer process from abstract middleware to actual middleware. For example, an application designed by using the MOM services provided by AME could be mapped onto a TupleSpace middleware such as TeenyLime.

5.2 TupleSpace to Database

This section describes the translation from tuplespace services to database services.

The TupleSpace paradigm uses a repository (the tuplespace) to store all the information: therefore the tuplespace contains tuples differing in number and type of their elements. A tuple T is an ordered set of elements

$T = \langle e_1, e_2, \dots, e_n \rangle$, where n represents the number of tuple elements. Before presenting the translation rule between tuplespace middleware and database middleware, let us partition the *tuple space* into subsets of homogeneous tuples; then a database table is created for each subset P_i as shown in Fig. 5.2.

The relationship between the *read* service of the tuplespace middleware and the query of the database middleware can be described as follows.

The WHERE condition has to be composed by the AND operation of the template actual fields, skipping the "wild cards" elements. Tuples can be also extracted from the tuple space using the destructive *take* operation; the corresponding implementation in a database middleware can be described as follows.

Finally, to *write* a tuple into the TupleSpace means to add information to our repository if it isn't there yet. In database programming paradigm, the same result

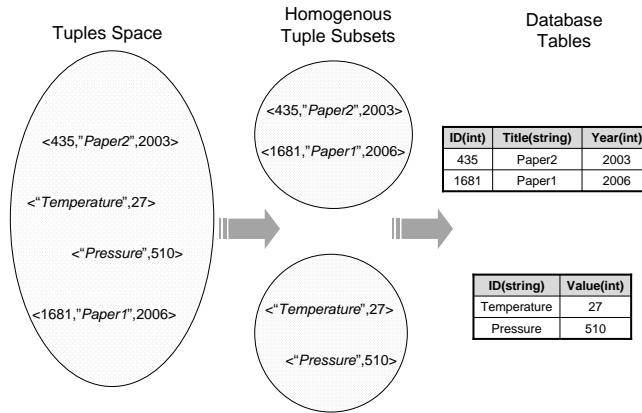


Fig. 5.2. Example of transformation of a tuple space into database tables.

```

read(<e1, e2, . . . , en>)
    ↓
SELECT *
FROM Pi
WHERE column1 = e1 AND
    . . .
    columnn = en

take(<e1, e2, . . . , en>)
    ↓
SELECT *
FROM Pi
WHERE column1 = e1 AND
    . . .
    columnn = en

DELETE
FROM Pi
WHERE column1 = e1 AND
    . . .
    columnn = en
    
```

is achieved by adding a line into the table of the tuple template. Therefore, a *write* operation is converted as follows. The *write* service inserts a tuple into the tuplespace only if that tuple does not already exist. The first operation to be performed then is a select query to check whether the table already contains that line or not. If the query finds the line, no operation is performed; otherwise a new line is added to the table with an insert query

```

write(<e1, e2, ..., en>)
↓
if ( SELECT *
      FROM Pi
      WHERE column1 = e1 AND
            ...
            columnn = en )
    // no operation
else
    INSERT ...

```

5.3 Database to Tuplespace

The translation between Database services into Tuplespace services is described in this section. The main objective of this translation is the creation of a tuple containing the same information typically stored through the database table. In order to create a database table inside the Tuplespace it's necessary to build two tuple including the table information (number of columns table and data type).

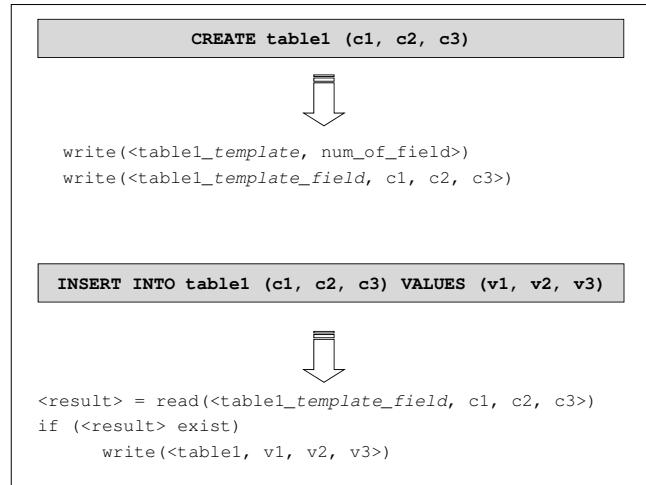


Fig. 5.3. Example of translation of a database into tuplespace representation.

Therefore, the `CREATE table1` query is translated with two `write` services to insert the following tuples (as shown in Figure 5.3):

- `< table1_template, num_of_field >` where `table1_template` is a keyword composed by the name of table to be created and the `”_template”` suffix and `num_of_field` is the number of columns table.

- $\langle table1_template_field, c1, c2, c3 \rangle$ where $table1_template_field$ is a keyword composed by the name of table to be created and the "_template_field" and $c1$, $c2$ and $c3$ are the column data type.

The `INSERT INTO table1` query is implemented in tuplespace programming paradigm by using a `read` and a `write` service to insert into the tuplespace an information corresponding to a new row table. Figure 5.3 reports the pseudo-code of this translation. It reads the tuple template ($read(\langle table1_template_field, c1, c2, c3 \rangle)$) to verify the presence of the corresponding table and then it writes a new tuple containing the row table data; this tuple is composed by a keyword representing the name of the table ($table1$) and by the values ($v1$, $v2$ and $v3$).

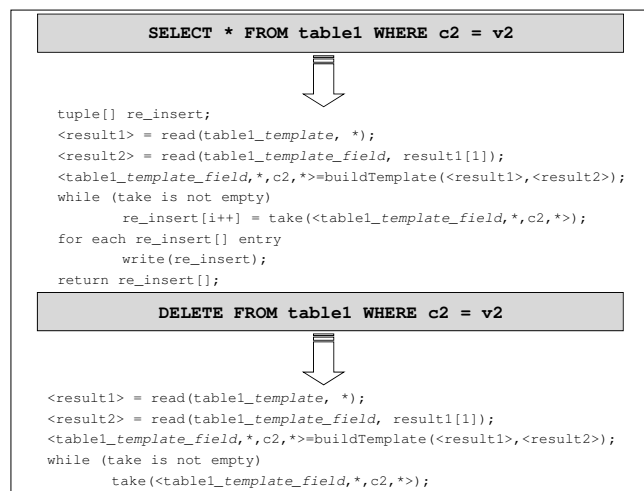


Fig. 5.4. Example of translation of a database into an tuplespace representation.

The `SELECT` query can be replaced inside the tuplespace programming paradigm by using the following tuplespace services:

- a couple of `read` service allows to know which table as to be used to select the information; the `buildTemplate` function builds the tuple template used to extract data from the table.
- a `while` statement allows to extract (by using a `take` service in order to avoid to extract the same tuple twice) the whole tuple set matching the tuple template previously created.
- finally, before to return the result (`return re_insert[]`), the set of tuples are re-inserted in the tuplespace.

Finally, the `DELETE` statement typically used to delete rows in a table is translated as the `SELECT` without re-insert operation in order to delete the tuples from the tuplespace.

The `SELECT` and `DELETE` statements are shown in Figure 5.4.

5.4 Object-oriented to Tuplespace

The Object-Oriented and Tuplespace paradigms are quite different. The former provides synchronous point-to-point communications between distributed objects. The latter has an asynchronous communication style and it uses a shared memory to link processes.

The lookup service of the Object-Oriented paradigm allows to obtain a local reference of the remote object. This reference is then used for method invocation. The same mechanism can be easily described with Tuplespace paradigm by writing and reading requests to/from the shared space.

Object-Oriented applications always obtain a reference to one particular object by using its public name given during registration. Therefore, in the Tuplespace implementation an identification string must be created in correspondence of a register operation:

$$REGISTER(*obj, name) \rightarrow stringID = name$$

The lookup service of the Object-Oriented paradigm allows to obtain a local object reference. Lookup is usually followed by a method invocation:

$$P = lookup(name) \\ R = P.METHOD_i(A_1, \dots, A_N)$$

Let us define client the object doing lookup and method invocation, and server the object implementing the actual object. To invoke one of the server methods, the client writes a tuple with the following fields:

- identification to the service provider (name)
- method to be called
- list of parameters belonging to the method signature.

For example, referring to the previous example, the client writes the following tuple:

$$\langle name, METHOD_i, A_1, \dots, A_N \rangle$$

The server, at the same time, is waiting for requests by performing a *take* operation with the following template:

$$\langle name, METHOD_i, *, \dots, * \rangle$$

where wild-cards correspond to parameters belonging to the method signature.

If a matching tuple is found, the server executes the method on the given parameters. Conversion from the Object-Oriented paradigm to the Tuplespace transforms methods into functions, though doing the same operations and with the same signature.

The result of the function is then returned by the server by writing a tuple with the following structure:

$$\langle "result", name, METHOD_i, A_1, \dots, A_N, R_i \rangle$$


where the first field allows to identify the response tuple which contains the returning value of the method.

This tuple is read by the client through a blocking *take* operation on the following template:

$$\langle \text{"result"}, \text{name}, \text{METHOD}_i, A_1, \dots, A_N, * \rangle$$

As a result, the register operation done in the OOM programming paradigm is replaced by an infinite loop of take operations, looking for method invocations, followed by write operations to publish results, as described in the following pseudo-code:

`register(obj, *name)`




```

while (1) {
  t = take(<name,*,*,...,*>);
  if (t.elementAt(0)==METHOD1) {
    P1 = t.elementAt(2);
    ...
    PN = t.elementAt(N);
    RET1 r1=obj.METHOD1(P1,...,PN);
    write(<"result",name,METHOD1,P1,...,PN,r1>);
  } else if {...}
  else if (t.elementAt(0)==METHODN) {
    P1 = t.elementAt(2);
    ...
    PN = t.elementAt(N);
    RETN rN=obj.METHODN(P1,...,PN);
    write(<"result", name,METHODN,P1,...,PN,rN>);
  }
}

```

The lookup and method invocations (performed by the client) are translated into a *write* operation, to publish the request, followed by a blocking *take* operation to get the result:

`P = lookup(name)`
`P.METHOD_i(P1,...,Pn)`



```

write(<name,METHODi,P1,...,PN>)
take
(<"result",name,METHODi,P1,...,PN,*>)

```

5.5 Tuplespace to Object-Oriented

The Tuplespace model uses a shared memory (called tuplespace), that usually contains tuples differing in number and type of their elements. Before mapping the Tuplespace paradigm on to the Object-Oriented one, the set of tuples must be partitioned into homogeneous subsets, based on equal number and type of fields. Then a class is created for the template of each subset, as shown in Figure 5.5.

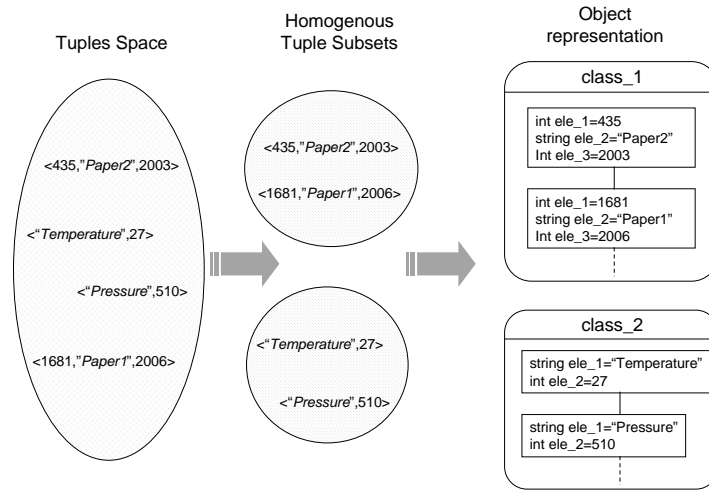


Fig. 5.5. Example of translation of a tuplespace into an object-oriented representation.

An instance object is created for each class. Since the tuplespace may contain more tuples of the same type, each instance object contains a list of nodes with as many fields as the tuple elements.

The translation between the *write* operation and the Object-Oriented operations is the following:

- when the first *write* is found, the corresponding Object-Oriented implementation has to create and register a new object of the corresponding class. The first node is added to the list and initialised with the tuple elements (step (1) of the following schema);
- further *write* operations are translated into invocations of the add method, with the tuple elements as parameters (step (2) of the following schema).

```
write(<e1, e2, ..., en>)
```



(1)

```
obj = new FileName();
register(*obj, className+counter());
obj.add(e1, e2, ..., en);
```

(2)

```
obj.add(e1, e2, ..., en);
```

The *read* and *take* services of the Tuplespace paradigm are based on a pattern-matching search. They return the first tuple corresponding to the provided template.

The *read* service is translated into the following Object-Oriented implementation:

- get a reference to each object (one a time) with the lookup service;
- invoke the "scan_list" method on the current object to check whether one node matches the template;
- if the "scan_list" method returns one node, it is returned as result;
- otherwise, the code passes to the following object;
- if no object contains a node matching the template, the service returns a null value.

```
read(<e1, e2, . . . , en>)
```



```
Node result = NULL;
for (int i=0; i<count; i++){
    obj=lookup("FileName"+i.str());
    Node n = obj;
    scan_list (<e1, . . . , en>);
    if (n!= null) {
        result=n;
        break;
    }
}
```

The translation of the *take* operation is similar:

- get a reference to each object (one a time) with the lookup service;
- invoke the "scan_list" method on the current object to check whether one node matches the template;
- if the "scan_list" method returns one node, it is removed from the list and returned as result;
- otherwise, the code passes to the following object;
- if no object contains a node matching the template, the service returns a null value.

Both the *read* and *take* operations require a search in the node list. To make the search faster, the actual implementation might use hash tables using tuple data as key values.

The resulting model is an Object-Oriented implementation but its communication style is still asynchronous and the search methods are based on pattern-matching operations. For this reason the application will not change its behaviour after the translation.

```
take(<e1, e2, ..., en>)
```



```
Node result = NULL;
for (int i=0; i<count; i++){
    obj=lookup("FileName"+i.str());
    Node n = obj;
    scan_list(<e1, ..., en>);
    if (n!= null) {
        remove_node(<e1, ..., en>);
        result=n;
        break;
    }
}
```

5.6 Tuplespace to Message-oriented

Before to describe the Tuplespace-to-MOM translation, let us define first how to translate the Tuplespace data (Tuples) into MOM data (Messages). Each tuple becomes a Topic object created by listing the tuple fields as a string. For instance, the following tuple:

< "Temperature" , 27, " Room1" , ID_SENSOR >

becomes:

"Temperature + 27 + Room1 + ID_SENSOR"

In the Tuple-space paradigm, each tuple is stored in the Tuple-space by using the write service. The subscribe service of the MOM paradigm inserts a record inside the AME table, still maintaining the relation between the subscribed topic and the event associated, as shown in Figure 3.2. Thus, the data representing a tuple can be stored inside a MOM middleware by using the subscribe service. Therefore, the Tuplespace write service is translated by using a MOM subscribe service as shown in Figure 5.6.

The implementation of the Tuplespace `read_nb` (non-blocking read) service inside the MOM leverages the use of topic object. The goal of the read service is to extract the tuple from the tuplespace, if it is present. In MOM this result can be reached in the following way:

- a NULL message is published in a particular topic describing the tuple (Topic object is created by listing the tuple fields as implemented for the write service);
- if the MOM contains this topic, the associated event is raised and the data event (containing the msg and the topic) is transmitted;
- the topic is translated in a tuple object and returned to the application invoking the read service.

The `read_b` (blocking read) is implemented exploiting the `read_nb`; this service is repeated until the tuple is retrieved.

This translation is shown in the pseudo-code of Figure 5.6.

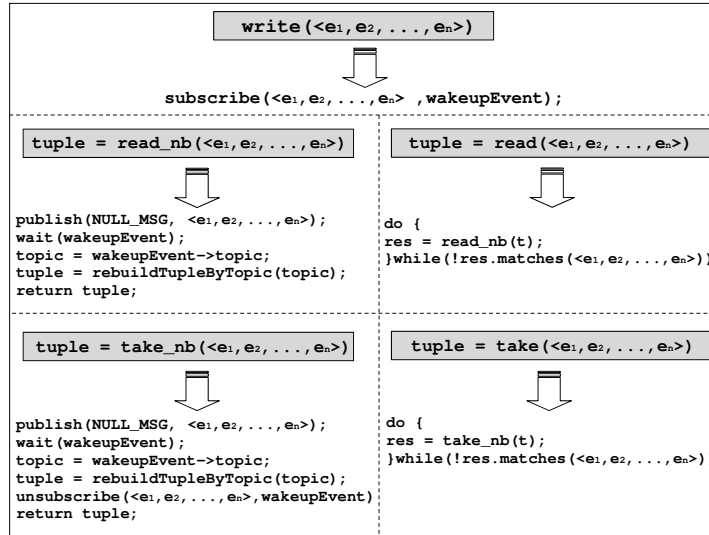


Fig. 5.6. Tuplespace to MOM.

5.7 Message-oriented to Tuplespace

Message-oriented middleware adopts a different communication paradigms with respect to the Tuplespace: the former offers an asynchronous model because the service’s consumers physically and temporally are decoupled from the service providers, while Tuplespace is a synchronous method

Let us define consumer the object doing subscribe, and producer the object calling the publish method. The publish service provided by the Message-oriented paradigm allows to store a message belonging to a particular topic into the message repository. This service can be easily performed by using the tuplespace middleware services writing in the tuples space a tuple keeping the following information:

- data information involved in the MOM message.
- name of the message topic.

This tuple is called tuple-data. For example, the MOM message *data* published by the producer in the message repository named *topic* can be transformed in the following Tuple:

$$\langle data, topic \rangle$$

The subscribe service allows to store a topic and an event that must be raised when a message with this topic will be published. This service can be performed by using the tuplespace middleware services by using a tuple (named tuple-events in the follow) including:

- list of events to be raised with respect to the topic.
- name of the message topic.

Therefore, the `publish(message, topic)` operation is replaced by a `write` operation (to store the message in the tuple-space) followed by a `read_nb` operation of tuple-events (to retrieve the list of the events belonging to the topic raised), as shown in Figure 5.7. The `subscribe(topic, event)` operation is implemented by the proxy-MW taking the tuple-event from the tuples space in order to update the list of events to be raised when a message belonging to the topic will be published; this translation is shown in the pseudo-code of Figure 5.7. The `unsubscribe(topic, event)` operation can be represented by using a `take` operation to retrieve the list of events previously defined through the `subscribe` operation. The event specified in the unsubscribe signature is deleted through this list of events (`deleteEventFromListEvent`) before updating this tuple containing the new list of events can be raised.

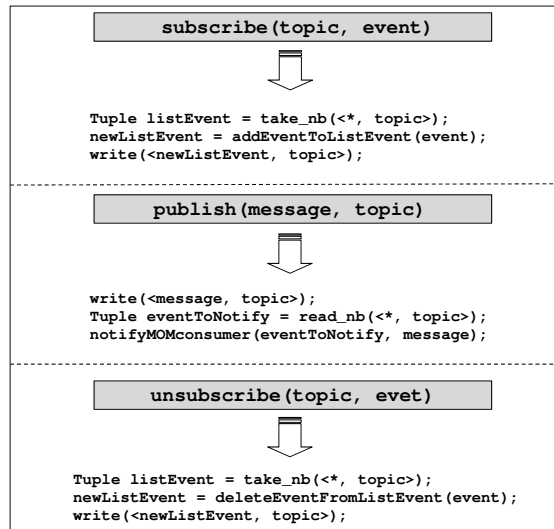


Fig. 5.7. MOM to TupleSpace.

5.8 Message-oriented to Database

This Section describes the translation from Message-oriented paradigm to Database paradigm. In the Message-oriented paradigm each entity interacts in an asynchronous way through the Publish/Subscribe paradigm enabling an event/notity mechanism. This mechanism is implemented through a data structure, named `event_repository`, to store the event used to automatically notify the entity subscriber when a message is published in that topic, as shown in Figure 3.2.

Dealing with data memorization aspects, the Database paradigm is powerful with respect to the Message-oriented paradigm, therefore this translation is not

complicated. The main concern regards the communication way: Database is query based (synchronous); on the other side Message-oriented is event/notify based (asynchronous).

The proxy middleware creates a table named `event_repository` exploiting the query service provided by the underlying middleware. This table keeps the same information as a typical `event_repository` structure would keep using directly a Message paradigm, that is an association between a Topic and an event for each entry.

Therefore, the `subscribe(topic, event)` operation is replaced by a `query` to insert into the DB the information related the event able to wake-up the entity associated with a particular topic. The `publish(message, topic)` operation is implemented by the proxy-MW selecting the information (the event list) related to a topic from the `event_repository` and raising the entities associated. Finally, the `unsubscribe(topic, event)` operation can be represented by using a `query` operation to delete the event from the `event_repository`. This translation is shown in the pseudo-code of Figure 5.8.

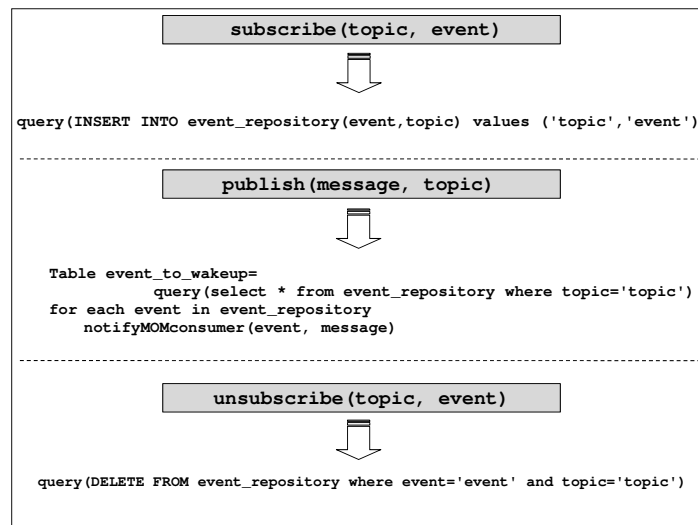


Fig. 5.8. MOM to Database.

5.9 Database to Message-oriented

The Database to Message-oriented translation can be performed by using the translation between Database to Tuplespace described in Section 5.3 and the translation between Tuplespace to Message-oriented presented in Section 5.6.

5.10 Message-oriented to Object-oriented

The Message-oriented to Object-oriented translation exploits the similarities between the two programming paradigm in term of data type saved inside the middleware. The object-oriented paradigm uses a `public repository` in which instances of the actual object have been registered with a public name (an entry of `<Reference, Object Name>`). The message-oriented paradigm memorizes an event and a topic inside the `event_repository` with the following entry: `<Event, Topic Name>`. Therefore the proxy-mw manages the `Reference` as a `Event` and the `Object Name` like a `Topic Name`.

The main issue is how to translate the asynchronous communication paradigm used by Message-oriented with respect to the synchronous communication paradigm implemented by Object-oriented. Moreover, the object-oriented paradigm requires each `Object Name` of a `Reference` object to be unique; in Message-oriented different communicating entities could be interested to the same topic. In order to solve this problem we have to make each entry unique by using a progressive number for the `Topic Name`.

For instance, these set of `subscribe` operations:

```
subscribe("Cartopic", event1);
subscribe("Shiptopic", event2);
subscribe("Cartopic", event3);
```

store the following entries:

```
< event1, "Cartopic1" >
< event2, "Shiptopic2" >
< event3, "Cartopic3" >
```

On the other hand, the following publish request:

```
publish(message, "Cartopic")
```

has to wake-up the following events:

```
< event1, "Cartopic1" >
< event3, "Cartopic3" >
```

Therefore, the `subscribe(topic, event)` operation is replaced by a `register` operation to insert an entry inside the OOM `public repository`. The `publish(message, topic)` operation is implemented by the proxy-MW looking for the entries (the event list) with a name formed by the topic required and a number from zero to the last number generated inside the `subscribe` and finally raising the entities associated. This translation is shown in the pseudo-code of Figure 5.9.

Concerning the `unsubscribe`, we note that the object-oriented paradigm doesn't provide a mechanism for data erasing. This problem is solved by registering another entry with the same name of the remote object to be erased with a keyword `NULL` corresponding to the event. Doing that when a `lookup` will be called to search this this object, a `NULL` event will be waken up (because the `lookup` does its research from the younger to the older entry).

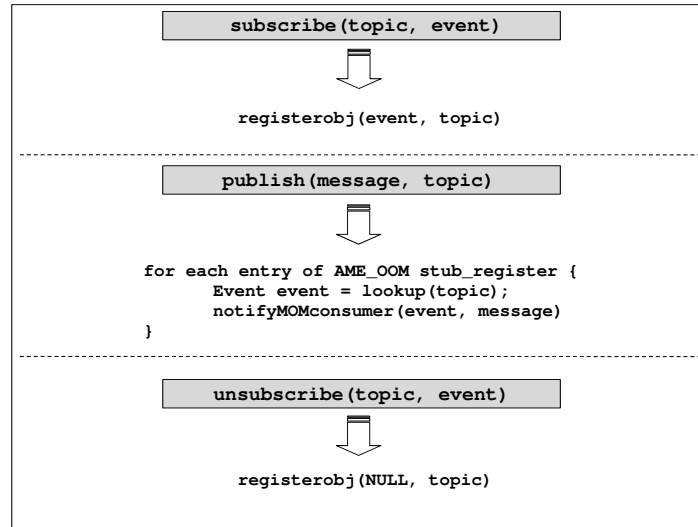


Fig. 5.9. MOM to OOM.

5.11 Object-oriented to Message-oriented

The Message-oriented paradigm uses a event based communication style (and is therefore asynchronous). The Object-oriented paradigm is fully synchronous. Moreover, the main difference between the two programming-paradigms consist of they have been designed in order to manage different contexts.

The Object-oriented to Message-oriented translation exploits the similarities between the two programming paradigm in term of data type saved inside the middleware, as already described in section 5.10.

In the Object-oriented paradigm, each reference to a particular object is stored in the public repository by using the `register` service. The subscribe service of the MOM paradigm inserts a record inside the AME table, still maintaining the relation between the subscribed topic and the event associated, as shown in Figure 3.2. Thus, the data representing a reference to the object can be stored inside a MOM middleware by using the subscribe service. Therefore, the Tuplespace `register` service is translated by using a MOM subscribe service as shown in Figure 5.10.

The `lookup` service of the Object-Oriented paradigm allows to obtain a local reference of the remote object. This reference is then used for method invocation. In MOM this result can be reached in the following way:

- a NULL message is published in a particular topic describing the public name given during registration;
- if the MOM contains this topic, the associated event is raised and the data event (containing the msg and the topic) is transmitted;
- the topic is translated in a reference object and returned to the application.

This translation is shown in the pseudo-code of Figure 5.10.

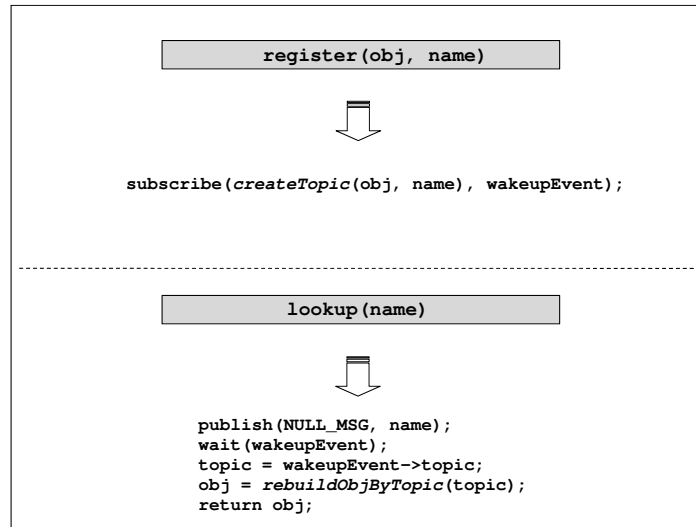


Fig. 5.10. OOM to MOM.

5.12 Object-oriented to Database

This Section explains the translation from Object-oriented to Database programming paradigm. Database and object-oriented paradigms have been designed to work in different context: the former focuses on the data store aspects; the second focuses on the communication aspects. Moreover, dealing with data memorization aspects, the Database paradigm is powerful with respect to the Message-oriented paradigm, therefore this translation is not complicated.

The translation is based on the creation of a table (called `Reference_Register`) inside the database. This table has to represent the OOM `public repository` in which instances of the actual object are registered with a public name (an entry of `<Reference, Object Name>`). Typically, the `Reference` is the address of an object; therefore, it can be easily kept inside the `Reference_Register` table as a string.

The translation can be implemented as shown in the pseudo-code of Figure 5.11.

5.13 Database to Object-oriented

The Database to Message-oriented translation can be performed by using the translation between Database to Tuplespace described in Section 5.3 and the translation between Tuplespace to Object-oriented presented in Section 5.5.

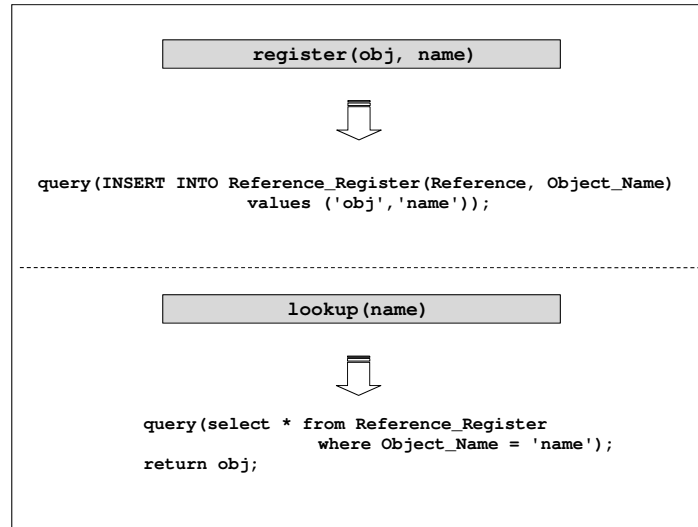


Fig. 5.11. OOM to Database.

5.14 Experimental analysis

The first step (AME3) of the proposed middleware-centric design flow has been applied to a house temperature monitor system scenario depicted in Figure 5.12. The objective is the evaluation of the translation mechanism described in this Section, pointing up the advantages of the proxy-MW solution.

In the House temperature monitor scenario a Wireless Sensors Network (WSN), including 20 nodes deployed in the house, sends ambient information to a remote service centre (SC) to control/monitor the house, through a fixed gateway installed in the house (GW). The gateway queries each temperature sensor and checks the received data to inform the remote service if some sample exceeds a given temperature threshold.

The reference application has been originally designed by using the AME programming paradigms: TupleSpace, Object-Oriented, Database, Message-oriented (in the follow called respectively TP-origin, OOM-origin, DB-origin and MOM-origin). The application models have been simulated in the AME environment to verify the correct implementation of functionalities.

Figure 5.13 shows SystemC code for the three actors of the application (WSN, GW, SC) modelled in TupleSpace programming paradigm by using AME environment. Figure 5.13.1 represents the application code running on each sensor node which samples the body temperature and then makes its value available by using the tupleSpace `write` service. Figure 5.13.2 represents the application code running on the GW which extracts (`take` service) available temperatures and checks for values above 40 degree; in this case, the GW generates an alarm through the `write` service. Figure 5.13.3 represents the application code running on the SC

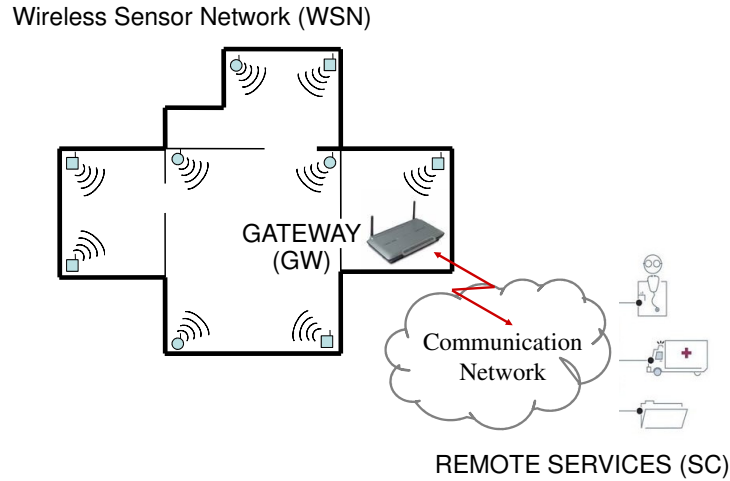


Fig. 5.12. House temperature monitor.

to verify whether an alarm has been raised (**take** service). Finally, Figure 5.13.4 describes the instantiation of the all actors.

Figure 5.14 reports the implementation of the house temperature monitor scenario, where the Database programming paradigm has been used and the specific API have been highlighted in bold face style. The code shows the daa exchanged beetwen the WSN, GW and SC by using the **query** service; in fact AME simulates the behaviour of a database. Figure 5.14.1 simulates the insertion of a set of records inside a **temper** table representing the temperature value monitored by the WSN. Figure 5.14.2 selects (by a **SELECT** query) these values, checks for values above 40 degree and generates an alarm in affermative case by inserting a reconrd inside a **alarms** table. Finally, Figure 5.14.3 selects the alarm records alarm to reise an alarm in dangerous situation.

Figure 5.15 and Figure 5.16 depict the pseudo-code of the house temperature monitor system scenario simulated by using the Object-oriented programming paradigm API provides by AME environment highlighted in bold face style.

Finally, the house temperature monitor scenario has been implemented by using Message-oriented programming paradigm as shown in Figure 5.17. It emphasises the use of the **publish** and **subscribe** services provided by AME message-oriented programming paradigm. The GW (Figure 5.17.2) is subscribed to the "temper" topic in order to receive the temperature data published by the WSN modules (Figure 5.17.1). When the GW catches a temperature value overcome 40 degree it generates an alarm publishing a message in the "alarm" topic. The SC is subscribed to the "alarm" topic; therefore, when the GW publishes an alarm the SC will be waken up to signal the alarm.

<pre> SC_MODULE(WSN) { // WSN.h ① sc_port<AME_if> wsn_port; void run(); SC_CTOR(WSN) : wsn_port("wsn_port"){ SC_THREAD(run); end_module(); } }; void WSN::run() { // WSN.cc while (1) { for (int i=0; i<NUM_SENSOR; i++){ Tuple t = ...<TEMP>...; wsn_port->write(t); } wait(); } } </pre>	<pre> SC_MODULE(GW) { // GW.h ② sc_port<AME_if> gw_port; void run(); SC_CTOR(GW) : gw_port("gw_port"){ SC_THREAD(run); end_module(); } }; void GW::run() { // GW.cc while (1) { Tuple = gw_port->take(...<TEMP>...); if (tuple[1]>40) gw_port->write(...<ALARM>...); wait(); } } </pre>
<pre> SC_MODULE(SC) { // SC.h ③ sc_port<AME_if> rs_port; void run(); SC_CTOR(SC) : rs_port("rs_port"){ SC_THREAD(run); end_module(); } }; void SC::run() { // SC.cc while (1) { rs_port->take(...<ALARM>...); wait(); } } </pre>	<pre> int sc_main() { ④ WSN *b = new WSN("WSN"); GW *gw = new GW("GW"); SC *rs = new SC("SC"); AME_3 *mw = new AME_3("AME_3"); b->wsn_port(mw->mw_port); gw->gw_port(mw->mw_port); rs->rs_port(mw->mw_port); sc_start(-1); return 0; }; </pre>

Fig. 5.13. Application described by using the AMS_3 library (Tuplespace).

Then, each original model has been translated applying the proxy-MW solution previously described. All translations between the different programming paradigms have been executed.

Table 5.1 reports simulation performance of the Tuplespace, Message-Oriented, Object-Oriented and Database application models, both the original and the translated versions. The first column reports the number of code lines of the application source code (hand-written in the case of original models and automatically-generated code in the case of translated models implemented by using the proxy-MW). The second column reports the simulation time; the third column reports the number of calls to AME services.

The results obviously show an increasing of the AME calls due to the translation mechanism. The Code lines parameter augments of a fixed value corresponding to the Code lines related to the proxy-MW used for the translation. For example, the proxy-MW used to translate a tuplespace application into a MOM application includes the pseudo-code described in Section 5.6 equal to 220 code lines.

Figure 5.18 shows the performance of AME proposed in this paper with respect to a solution using a parser/translator of SystemC code, which was implemented without the AME proxy-MW (called AMEplain). The comparison has been implemented for the Tuplespace-to-TOM translation, because it uses the proxy-MW more extensive in terms of Code lines (220 code lines as reported in Table 5.1).

The comparison has been implemented for an application originally written by using the Tuplespace paradigm. It is composed by 500 code lines. The results clearly show that the translation mechanism is application-dependent. The real advantage by using the proxy-MW takes place when the code of the original

<pre> SC_MODULE(WSN) { // WSN.h sc_port<AME_if> wsn_port; void run(); SC_CTOR(WSN):wsn_port("wsn_port"){ SC_THREAD(run); end_module(); } }; void WSN::run(){ // WSN.cc while (1) { for (int i=0;i<NUM_SENSOR;i++){ String sql="insert into temper values(..+<TEMP>+..)"; wsn_port->query(sql); } wait(); } } </pre>	<pre> SC_MODULE(GW) { // GW.h sc_port<AME_if> gw_port; void run(); SC_CTOR(GW) : gw_port("gw_port"){ SC_THREAD(run); end_module(); } }; void GW::run() { // GW.cc while (1) { String sql="select * from temper;"; result=gw_port->query(sql); line=result.get_line(); if (line[1]>40){ String sql="insert into alarms values(..+<ALARM>+..)"; gw_port->query(sql); } wait(); } } </pre>
<pre> SC_MODULE(SC) { // SC.h sc_port<AME_if> rs_port; void run(); SC_CTOR(SC) : rs_port("rs_port"){ SC_THREAD(run); end_module(); } }; void SC::run() { // SC.cc while (1) { String sql="select * from alarms;"; msg = rs_port->query(sql); if (msg == <ALARM>) { sql="delete from alarms where..;"; rs_port->query(sql); } wait(); } } </pre>	<pre> int sc_main() { WSN *b = new WSN("WSN"); GW *gw = new GW("GW"); SC *rs = new SC("SC"); AMS_3 *mw=new AMS_3("AME_3"); b->wsn_port(mw->mw_port); gw->gw_port(mw->mw_port); rs->rs_port(mw->mw_port); sc_start(-1); return 0; }; </pre>

Fig. 5.14. Application described by using the AMS_3 library (Database).

	Code lines	Simulation time [msec.]	AME calls
TP-original	X	352	5442
TP_2_MOM	X+220	3288	3595
TP_2_OOM	X+100	235	6595
TP_2_DB	X+90	5070	12896
MOM-original	Y	88	2086
MOM_2_TP	Y+50	344	4172
MOM_2_DB	Y+30	436	2087
MOM_2_OOM	Y+35	58	4170
OOM-original	Z	40	3321
OO_2_TP	Z+90	108	3343
OO_2_MOM	Z+140	776	3343
OO_2_DB	Z+40	796	3344
DB-original	W	1684	2252
DB_2_TP	W+120	319	4675
DB_2_OOM	W+100	121	4202
DB_2_MOM	W+340	879	8270

Table 5.1. Simulation performance results of the programming paradigms translations.

application includes a high number of AME calls (X-axis) since it is not necessary to replace the translation-paradigm code (as in AME-plain) in each point of the original code. In fact, when the AME calls increases, the translated code (Y-axis) is a constant value equal to the original application (500 lines) added

<pre> // WSN.h SC_MODULE(WSN) { sc_port<AME_if> wsn_port; int id; void run(); SC_CTOR(WSN):wsn_port("wsn_port"){ SC_THREAD(run); end_module(); } }; // WSN.cc void WSN::run(){ wsn_port->register(&remote,id); } </pre> <hr style="border-top: 1px dashed black;"/> <pre> // WSN_IF.h class wsn_if: public remote_generic{ virtual int getTemperature(); }; // WSN_CLASS.h class wsn_class: public wsn_if{ int getTemperature(); }; // WSN_CLASS.cc int wsn_class::getTemperature(){ return <TEMP>; } </pre>	<pre> // GW.h SC_MODULE(GW) { sc_port<AME_if> gw_port; void run(); SC_CTOR(GW) : gw_port("gw_port"){ SC_THREAD(run); end_module(); } }; // GW.cc void GW::run() { while (1) { for (int i=0;i<NUM_SENSOR;i++){ bsn_class ref=lookup(i); int t= ref.getTemperature(); if (t>40) sc_class ref=lookup("remote"); remote.setAlarm(<ALARM>); } wait(); } } </pre>
--	--

Fig. 5.15. Application described by using the AMS_3 library (Object-oriented) - PART 1.

to the Tuplespace-to-MOM proxy-MW (equal to 220 lines). Moreover, the AME proxy-MW solution outperforms the AME-plain solution decreasing the execution time of the translated application (3288 msec. vs. 10589 msec.).

<pre> // SC.h SC_MODULE(SC) { sc_port<AME_if> rs_port; remote_class remote; void run(); SC_CTOR(SC) : rs_port("rs_port"){ SC_THREAD(run); end_module(); } }; // SC.cc void RS::run() { rs_port->register(remote,id); } </pre> <hr/> <pre> // SC_IF.h class sc_if: public remote_generic{ virtual void setAlarm(string alarm); }; // SC_CLASS.h class sc_class: public rs_if{ void setAlarm(string alarm); }; // SC_CLASS.cc void sc_class::setAlarm(string alarm){ cout<<alarm<<endl; } </pre>	<pre> int sc_main() { WSN *b = new WSN("WSN"); GW *gw = new GW("GW"); SC *rs = new SC("SC"); AME_3 *mw=new AME_3("AME_3"); b->wsn_port(mw->mw_port); gw->gw_port(mw->mw_port); rs->rs_port(mw->mw_port); sc_start(-1); return 0; }; </pre>
---	---

Fig. 5.16. Application described by using the AMS_3 library (Object-oriented) - PART 2.

<pre> SC_MODULE(WSN) { // WSN.h sc_port<AME_if> wsn_port; void run(); SC_CTOR(WSN) :wsn_port("wsn_port"){ SC_THREAD(run); end_module(); } }; void WSN::run() { // WSN.cc while (1) { for (int i=0;i<NUM_SENSOR;i++){ int t=TEMPERATURE; wsn_port->publish(t,"temper"); } wait(); } } </pre>	<pre> SC_MODULE(GW) { // GW.h sc_port<AME_if> gw_port; void run(); void notify_routine(); SC_CTOR(GW) : gw_port("gw_port"){ SC_METHOD(run); SC_THREAD(notify_routine); sensitive << wakeup_gw; end_module(); } }; void GW::run() { // GW.cc gw_port->subscribe("temper",wakeup_gw); }; void GW::notify_routine() { // GW.cc while(1) { wait(wakeup_gw); int temp = wakeup_gw.getValue(); if (temp > 40) gw_port->publish(temp,"alarm"); } }; </pre>
<pre> SC_MODULE(SC) { // SC.h sc_port<AME_if> rs_port; void init(); void run(); SC_CTOR(SC) : rs_port("rs_port"){ SC_METHOD(init); SC_THREAD(run); sensitive << wakeup_rs; end_module(); } }; void SC::run() { // SC.cc rs_port->subscribe("alarm",wakeup_rs); } void SC::init() { // SC.cc while(1) { wait(wakeup_rs); cout<<ALARM<<wakeup_rs.getValue(); } } </pre>	<pre> int sc_main() { WSN *b = new WSN("WSN"); GW *gw = new GW("GW"); SC *rs = new RS("SC"); AME_3 *mw=new AME_3("AME_3"); b->wsn_port(mw->mw_port); gw->gw_port(mw->mw_port); rs->rs_port(mw->mw_port); sc_start(-1); return 0; }; </pre>

Fig. 5.17. Application described by using the AMS_3 library (Message-oriented).

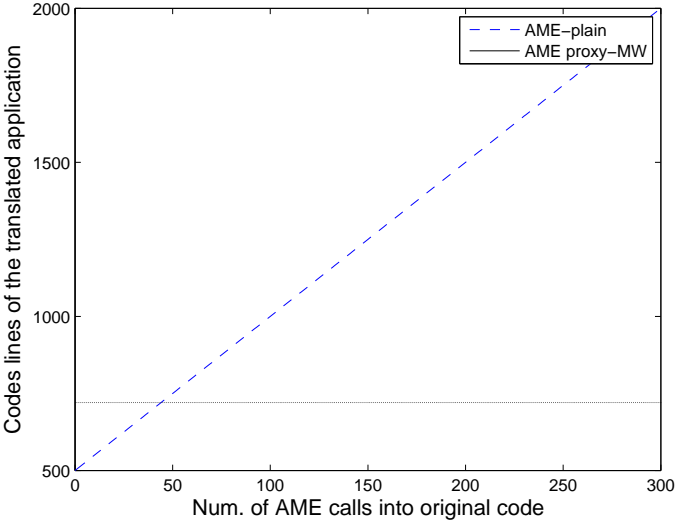


Fig. 5.18. AME proxy-MW vs. AME plain.

Mapping

As already described in Section 1, the key aspects of NES applications are their distributed nature and the presence of very limited HW resources, as in case of Wireless Sensor Network (WSNs). The wide adoption of these applications requires the interoperability across different manufacturers, the simplification of application development through abstract paradigms, simulation tools to verify the correct behavior and the fulfilment of the tight HW/SW constraints.

Interoperability is achieved through the use of standard protocol stacks (e.g., IEEE 802.15.1/Bluetooth and IEEE 802.15.4/ZigBee [74, 75]). For instance, the ZigBee standard provides the so-called *Profiles* which define services and attributes of nodes implementing common applications (e.g., for home automation).

Simplification of application development can be achieved through software services provided by the operating system. TinyOS is a popular operating system providing a component-based programming approach [76]. SW services can also be implemented by a *middleware*, which hides the peculiarities of operating system and HW components and provides abstract entities (such as objects, tables, tuples and message boards) [35, 77]. The Texas Instruments' Z-Stack [78] is a software package providing ZigBee API and minimal OS services, thus behaving as a middleware.

Simulation tools are used for validating the application: they range from pure network tools, such as NS-2 [53], to platform-specific environments, such as TOSSIM [47].

The integration of these three aspects is the objective of Abstract Middleware Environment (AME) in order to provide a complete design methodology for NES applications. The proposed methodology allows programmers to write NES applications by using AME framework for fast simulation and validation as described in Section 5 and 4. AME behaves as an abstraction of the services provided by the actual platform. Finally, the implemented application has to be automatically mapped over an actual platform. This design step is called AME *Mapping* process and it regards the deployment of the application over the actual NES.

Figure 6.1 shows an example where the Mapping of the AME-based application is applied onto a typical middleware for WSN, called ZigBee.

This Chapter completes the description of AME design methodology by automatically mapping AME applications over a target NES platform, e.g., the Texas

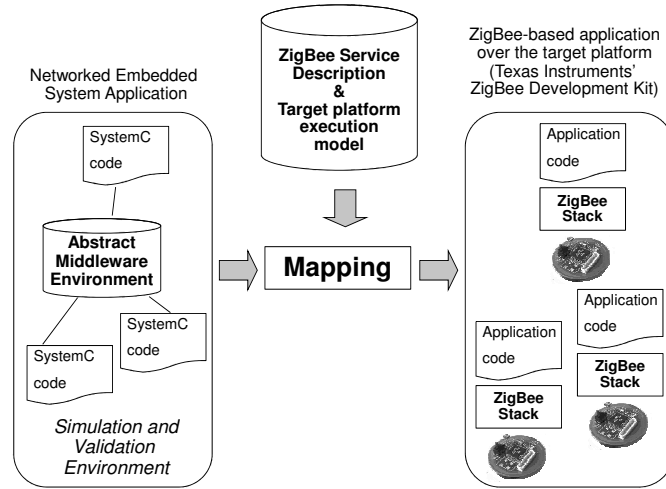


Fig. 6.1. Mapping of the AME-based application onto ZigBee.

Instruments CC2430 nodes, running Z-Stack middleware (ZigBee), which is complex and generic enough to verify the effectiveness of the methodology. Moreover, experimental results are reported to show the advantages of the AME-centric design methodology. Finally, the *Mapping* process onto a tuplespace programming paradigm (named TeenyLime) is described.

6.1 Mapping onto ZigBee/Z-Stack

6.1.1 Object-Oriented AME

To describe the mapping process from the AME application to the ZigBee application we chose the object-oriented programming paradigm for its spread among programmers. A typical object-oriented middleware [35, 78] provides: a mechanism to describe an object interface and to map it onto an actual object; a public repository in which instances of actual objects are registered, so that a client can obtain a local reference of a remote object; a protocol to remotely invoke object's methods with transmission of parameters and results. The object-oriented services provided by AME have been described in Section 3.2.3.

To better clarify the use of AME with the object-oriented paradigm, let us consider the home automation scenario depicted in Figure 6.2 where a switch controls a lamp through a wireless channel.

Figure 6.3 reports the corresponding application code described in SystemC with the Abstract Middleware Environment.

Figure 6.3.1 reports the SystemC main function which creates the `light` and the `switch` objects communicating each other through an instance of the AME middleware. The `OnOffCluster` is an abstract class which defines the commands

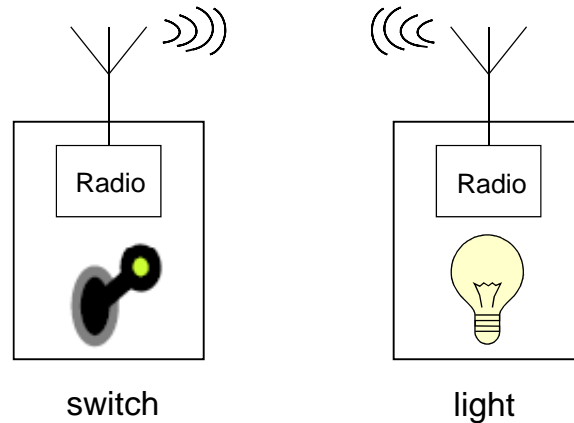


Fig. 6.2. Wireless application for light control.

(`COMMAND_OFF`, `COMMAND_ON` and `COMMAND_TOGGLE`), the attribute `OnOff` (i.e., the light status) and the operation (`OnOffCB`) which can be performed on the attribute. This class represents the description of a functionality, i.e., an interface in object-oriented terms; in Section 6.1.2 we will see it is quite close to the concept of *cluster* in ZigBee standard. Figure 6.3.2 represents `HomeAutomation` object implementing the `OnOffCB` function that modifies the light status according to the commands defined in the implemented interface `OnOff.Cluster`:

- `COMMAND_ON`: the light is switched on;
- `COMMAND_OFF`: the light is switched off;
- `COMMAND_TOGGLE`: to change light status.

Figure 6.3.3 and Figure 6.3.4 describe two modules representing the actors of the distributed application. Figure 6.3.3 describes the object named `light`; during initialization (`init()` method) it calls the AME `registerobj()` service to register an instance of the `HomeAutomation` object so that its methods can be remotely accessed. Figure 6.3.4 describes the object named `switch`; during initialization (`init()` method) it retrieves a reference to the instance running on the other node by using the AME `lookup()` service. During the execution phase (`run()` method), the object calls `OnoffCB()` method on the remote object to change light status. It is worth to note that only the `switch` object defines the `run()` method and a corresponding thread while the `light` object plays a passive role since it provides a remote object whose methods are executed in the middleware thread.

The same application can be modelled in AME framework at AME.2 level involving a simulated wireless channel. Figure 6.4 reports the AME.2 architecture where the NES application described in Figure 6.3 is simulated taking in account wireless network effects.

Figure 6.5 shows the SystemC pseudo-code to model the scenario at AME.2 design level; `light` and `switch` modules instantiated at line 3 and 4 remain unchanged with respect to the AME.3 implementation described in Figure 6.3. Lines

<pre>/* main.cc */ int sc_main(int argc, char *argv[]) { AME *mw = new AME("Middleware"); light *l_board=new light("LIGHT"); swtch *s_board=new swtch("SWITCH"); l_board->mw_port(mw->oom_port); s_board->mw_port(mw->oom_port); sc_start(-1); return 0; };</pre>	<pre>/* OnOff_Cluster.h */ //COMMANDS CONSTANTS #define COMMAND_OFF 0 #define COMMAND_ON 1 #define COMMAND_TOGGLE 2 class OnOff_Cluster: public remote{ //ATTRIBUTES bool OnOff; //SERVER CALLBACKS virtual void OnOffCB(int com)=0; };</pre>
<pre>/* HomeAutomation.h */ class HA : public OnOff_Cluster { public: void init(); void OnOffCB(int command); };</pre>	<pre>/*HomeAutomation.cc*/ void HA::OnOffCB(int command){ if (command == COMMAND_OFF){ OnOff = false; }else if (command==COMMAND_ON){ OnOff = true; }else if (command==COMMAND_TOGGLE){ if (OnOff == false) OnOff = true; else OnOff = false; } };</pre>
<pre>/*light.h*/ SC_MODULE(light) { sc_port<mw_oom_if> mw_port; bulb led; void init(); SC_CTOR(light) { SC_THREAD(init); } };</pre>	<pre>/* light.cc */ void light::init(){ if(!mw_port->registerobj(&led, "LGT") { cout<<"Light not present"<<endl; } cout <<"Register executed!"<< endl; };</pre>
<pre>/* switch.h */ SC_MODULE(switch) { OnOff_Cluster *l; sc_port<mw_oom_if> mw_port; void init(); void run(); SC_CTOR(switch) { SC_THREAD(init); SC_THREAD(run); } };</pre>	<pre>/* switch.cc */ void switch::init(){ do { l=(OnOff_Cluster*) mw_port->lookup("LGT"); } while (l == NULL) }; void switch::run() { while(1){ if (l!=NULL) l->OnOffCB(COMMAND_TOGGLE); } };</pre>

Fig. 6.3. Light-control application designed by using the AME-3.

6 and 7 emphasize the instantiation of one AME_2 module and line 6 and 7 report the creation of the connection between the NES applications and the AME middleware.

Lines 11-24 show the insertion of the SCNSL-AME-Transactor for each AME_2 module previously created; this module allows to connect SCNSL network simulator (Line 26) simulating IEEE 802.15.4 protocol to AME_2.

The example shows that object-oriented programming and SystemC concurrency model (i.e., threads) allow to describe efficiently a distributed application and, therefore, the AME-based approach can save design and coding effort.

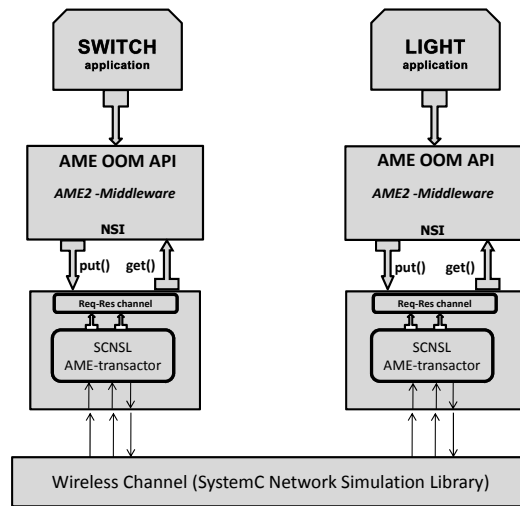


Fig. 6.4. Light-control architecture designed by using the AME.2.

6.1.2 ZigBee

ZigBee is a multi-vendor standard for low-power, low-data-rate, low-cost wireless communications enabling next-generation sensor and actuation networks [75]. ZigBee is defined on top of IEEE 802.15.4 [74] and it adds the capability of creating interoperable applications over multi-hop networks. The key for the interoperation between devices on a ZigBee network is the agreement on a profile. *ZigBee Application Profiles* are standard agreements on type of messages, message formats and processing actions that allow to announce, query and discover node capabilities. For example, if the two nodes depicted in Figure 6.2 are compliant to the ZigBee standard and, in particular, to the *Home Automation* profile, then they can recognize each other's functionality (i.e., lamp and switch) and the actions one node can perform on the other node (e.g., changing light status).

In ZigBee standard, message formats are described as *clusters* and identified with a code which is unique within a given Application Profile. Clusters can be groups of attributes or commands and the corresponding Application Profile defines the valid match between attributes and commands; for example, in the Home Automation Profile, a lamp can be recognized for its specific set of commands, namely those related to light switching, while a switch can be recognized for each attributes, namely the status, and lamp-switch connections are valid. Clusters are quite similar to objects in the object-oriented programming paradigm and this fact suggests a way to efficiently map the previously described object-oriented AME applications onto ZigBee.


```

1 int sc_main(int argc, char *argv[]){
2 // application implementation
3 light *l_board=new light("LIGHT");
4 swtch *s_board=new swtch("SWITCH");
5 // Middleware creation
6 AME_2 *mw_light=new AME_2("AME_2");
7 AME_2 *mw_switch=new AME_2("AME_2");
8 // Connection between Application and AME2
9 l_board->mw_port(mw_light->oom_port);
10 s_board->mw_port(mw_switch->oom_port);
11 // Request and response channel creation
12 ReqResp *req_rsp [2]=new ReqResp("channel")
13 // Connection between AME2 and request/response channel
14 mw_light->net_out(req_rsp[0]->put_request_export);
15 mw_light->net_in(req_rsp[1]->get_request_export);
16 mw_switch->net_out(req_rsp[2]->put_request_export);
17 mw_switch->net_in(req_rsp[3]->get_request_export);
18 // SCNSL-AME-Transactor creation
19 SCNSL-AME_trans *transactor[2]=new SCNSL-AME_trans("trans")
20 // Connection between request/response channel and SCNSL-AME-Transactor
21 transactor->net_out(req_rsp[0]->put_request_export);
22 transactor->net_in(req_rsp[1]->get_request_export);
23 transactor->net_out(req_rsp[2]->put_request_export);
24 transactor->net_in(req_rsp[3]->get_request_export);
25 // SCNSL simulator (IEEE 802_15_4)
26 SCNSL *network = new SCNSL("wnet")
27 network->setClock(clock)
28 foreach transactor in transactor[]
29     transactor->setMacNode(network->802_15_4mac)
30 ...
31 sc_start(-1);
32 return 0;
33 };

```

Fig. 6.5. Light-control pseudo-code designed by using the AME.2.

6.1.3 Z-Stack Execution Model

The mapping of an AME-based application onto a given HW/SW platform requires to take into account the execution model of the target operating system, i.e., how it handles tasks, threads, events and so forth. AME-based applications must be adapted to move from the execution model provided by SystemC to the one provided by the target platform. In particular, in our case study, we adopted Texas Instruments' CC2430 nodes which use the Z-Stack operating system [78]. This platform represents one of the most popular and mature implementation of ZigBee standard and its constraints in terms of memory usage allows to show the potentialities of the approach. Z-Stack provides a full ZigBee stack and the minimal services provided by the Operating System Abstraction Layer (OSAL), i.e., inter- and intra-task communications task scheduling, timer management, and dynamic memory allocation. Furthermore, every application can use the HW Abstraction Layer API to control HW components like LEDs, displays, etc.

The structure of a typical Z-Stack application is as follows:

- *Task Initialization* (`Init()` method): it performs initialization of data structures that describe the status of the node.
- *Task Event Handler* (`ProcessEvent()` method): the ZigBee application follows an event-driven execution model, i.e., it performs operations when certain events occur. This method receives events and determines which operations have to be performed. While handling some events is mandatory (e.g., `SYS_EVENT_MSG`); the user can define other events, e.g., a timeout.

- *Other methods*: callbacks to be executed when certain events occurs, methods to send messages and other functions.

6.1.4 Methodology

This Section describes the process to map AME-based applications onto the CC2430 SoC by Texas Instruments. The mapping process consists of three key concepts:

- *Compliance with the Z-Stack execution model*: to correctly interact with the Z-Stack operating system, the translated application has to use the same methods as typical Z-Stack applications, i.e., the task initialization method and an event listener (Task Event Handler) as described in Section 6.1.3. In this way, the application will be able to receive events from lower layers and thus from other nodes. Section 6.1.4 describes this issue.
- *ZigBee OOM Services implementation*: calls to object-oriented AME services described in Section 6.1.1 have to be translated to work onto ZigBee stack. If the involved objects belong to an existent ZigBee Application Profile, then mapping is simplified, otherwise additional functions have to be implemented on top of the ZigBee stack. This issue is addressed in Section 6.1.4.
- *SystemC to C language conversion*: AME applications are written in SystemC (i.e., C++), while Z-Stack applications are written in C language. Conversion has to take into account not only language differences but also the issues described in the previous two points. The conversion is performed with the support of an automatic tool; this issue is described in Section 6.1.4.

Compliance with Z-Stack

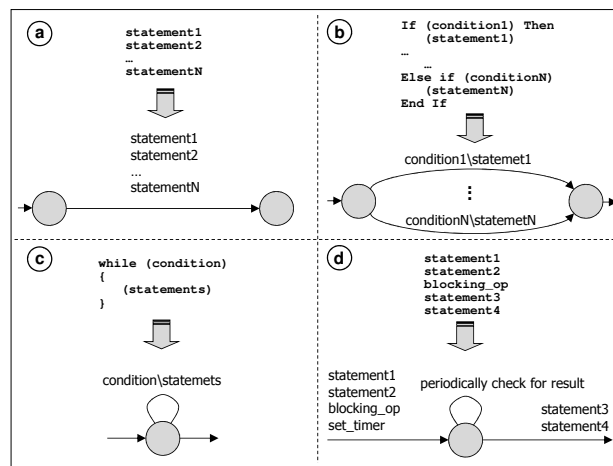


Fig. 6.6. Basic cases for the generation of the application FSM.

At a first look, it seems straightforward to map AME code onto a Z-Stack application; initialization code can be put in the `Init()` function while the core of the application code can be put in the `ProcessEvent()` function. Actually a problem arises with blocking calls such as the `lookup()` service and remote method invocations which need to stop the calling process until a response arrives. The Z-Stack operating system does not support pre-emption and therefore blocking calls in application code block the event listener thus preventing the correct processing of system events.

To adapt application code to the Z-Stack execution model, the blocking calls must be replaced by their non-blocking versions plus a timer initialization which gives control to the OS and periodically polls for results. To perform automatic code adaptation we decided to rely on code representation through finite state machine (FSM). The FSM is implemented inside the `ProcessEvent()` method as shown in Figure 6.8. The FSM is scheduled by using a timer; when the timer expires then a `TIMER_EVENT` is generated by the OS and the FSM re-starts from the last state reached (the timer is re-started after every expiration). The `ProcessEvent()` function also manages the arrival of network messages by catching the event named `MESSAGE_EVENT` generated by the OS, as shown in Figure 6.8.

The core of the AME code (i.e., statements to be put in the body of the `ProcessEvent()`) is converted into an FSM by applying recursively the cases depicted in Figure 6.6. As shown in Case A, non-blocking statements are put on FSM transitions while branches are implemented through multiple transitions (Case B). Loops are translated into self-looping states (Case C). Sections containing blocking operations undergo a more complex translation (Case D). A new state is created and statements preceding the blocking call are put of the entering transition together with the non-blocking version of the call and a timer initialization. Then, application remains in the new state until a response arrives. Finally, remaining statements are put on the transition leaving the described state. Thanks to the FSM formal representation, the described conversion rules can be performed by an automatic tool based on syntax checking; furthermore, rules can be recursively applied in more complex cases (e.g., a blocking call inside a loop).

ZigBee OOM Services

As described in Section 6.1.1, object-oriented AME provides a mechanism to describe an object interface and to map it onto an actual object, a mechanism to obtain a local reference of a remote object, and a public repository in which instances of the actual objects are registered.

Once the application becomes a really distributed application, object references used in the AME environment have to be replaced by the addresses of the nodes where the objects are. Therefore, the public repository is needed to link the public name of an object and its node location; two solutions are available:

- every node has a copy of the public repository;
- only the ZigBee network coordinator keeps the public repository.

The second design choice allows to maintain public repository information consistent and updated. If a node joins the WSN during the normal network operations

(hot plug-in), it must get a complete version of the table, including information that has been exchanged when it was not part of the network yet. Moreover, in a typical WSN the coordinator has more memory resources with respect to the other wireless nodes, and therefore it is able to maintain the whole public repository.

The AME services, described in Section 6.1.1, are implemented on the ZigBee stack as described in the following. A pictorial representation of data exchanges between nodes is given in Figure 6.7.

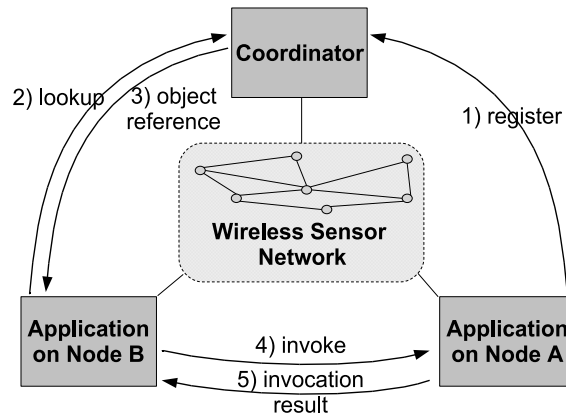


Fig. 6.7. ZigBee implementation of OOM services.

The *register service* is implemented by sending a message to the coordinator with the object public name, node address and a field that indicates that this is a register request. On receipt, the coordinator adds the new entry to the public repository: in this way the information becomes available to the other nodes.

The *lookup service* is implemented by sending a lookup message containing the public name of the remote object and a field that indicates that this is a lookup request. On receipt, the coordinator sends back a message containing the address of the node. In this way, the caller obtains the node address and can interact directly with it.

The implementation of remote method invocation depends on the support of ZigBee Profiles by the target platform. Therefore two mechanisms are possible.

Non-profile-enabled platform: a remote method invocation is converted as follows:

- Node A, that wants to invoke the method on node B, sends a message to B containing the name of the method to be executed and the parameters, plus a field indicating that this is a method invocation message.
- On receipt, node B executes the method.
- Node B sends to A a message containing the values received and the execution result, plus a field indicating that this is a method execution result message.
- On receipt, node A obtains the result of its invocation and the application flow goes on.

Profile-enabled platform: If the target platform supports Profiles then the application can use some advanced communication mechanisms (such as commands and callbacks defined by the specific profile) and a precise protocol for the communication. The key element in profile-enabled applications is the *cluster* which embeds attributes and commands related to a specific real-world object (e.g., a lamp or a switch). If method invocations refer to objects supported by the profile then they can be implemented by native profile mechanisms without the need of additional code over the ZigBee stack.

In the light-control example presented in Section 6.1.1, the switch node obtains a reference to the remote object and then invoke its `OnOff` method, with the right parameter to switch the light on. The same behavior can be obtained by using ZigBee messages contained in the Home Automation Profile.

SystemC to C language conversion

Figure 6.8 shows the operations performed to transform the AME application (written in SystemC language) into a Z-Stack compliant application (written in C language).

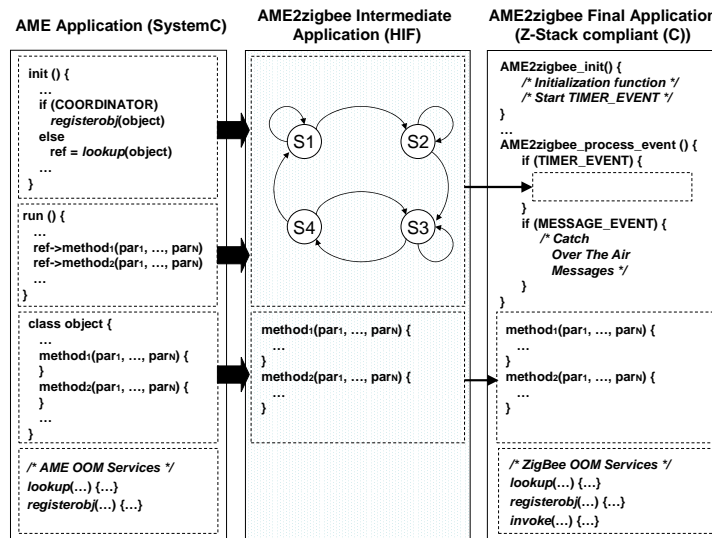


Fig. 6.8. AME to Z-Stack mapping process.

The mapping process consists of three main steps:

1. During the first phase, the AME application is automatically translated into an intermediate format which captures the basic syntax elements of the source code (i.e., sequential statement blocks, branches and function calls).
2. In the second step, the intermediate representation is manipulated to:
 - separate initialization statements from application statements;

- extract the application FSM by identifying flow branches and blocking statements as described in Section 6.1.4;
 - identify calls to OOM services which must be translated as described in Section 6.1.4;
 - identify remote method invocations which must be handled differently from local function calls.
3. Finally, the intermediate representation is automatically converted into a C-language application which can be compiled and downloaded on the target platform.

To simplify conversion, objects to be mapped on ZigBee applications (e.g., `switch` and `light` in the light-control example) have to be written by respecting some guidelines. In particular, to simplify the separation between initialization code and application code, objects must provide the following methods:

- `init()`: it contains the operations that should be performed just once, such as variable initializations, object register and object lookup operations;
- `run()`: it contains the body of the application; typically, in SystemC this portion of code is placed into the `SC_THREAD` process;
- other application-specific methods.

The AME Automatic Mapping tool has been implemented by using HIF Suite [84]. HIF stands for *HDL Intermediate Format* and it is an XML language that allows tree-structured descriptions of HW/SW objects. Each object describes a specific functionality or component that is typically provided by HDL languages like SystemC. The HIF Suite is composed of a set of tools and functions, based on the HIF language, that allow system designers to:

- parse SystemC functional descriptions
- extract an HIF representation of the descriptions
- manipulate the HIF representation through a set of functions
- generate a C-language description that reflects the changes introduced by the manipulation of the HIF representation.

6.1.5 Experimental analysis

The proposed SystemC-centric approach has been evaluated by considering two different ZigBee applications contained in the Texas Instruments' Z-Stack distribution, i.e., `GenericApp` and `HomeAutomation`. The former establishes a connection between two nodes and periodically exchanges a string between them; the latter implements the light-control application described in Section 6.1.1. The former application does not use any ZigBee Application Profile while the latter uses the Home Automation Profile: this difference allows to show the effect of Profiles on the mapping performance.

The functional behavior of the applications has been used to write the corresponding AME-based applications. Table 6.1 evaluates the programming efficiency achievable by the use of SystemC, i.e., the resulting simplification of the programming task. The complexity of the source code of both the AME-based applications is compared to the original applications. The first column reports the number of

	Language	Code lines	Num. API calls
AME_GenericApp	SystemC	219	3
GenericApp	C	565	15
AME_HomeAutomation	SystemC	178	2
HomeAutomation	C	720	12

Table 6.1. Results on programming efficiency.

	Code lines	Code size (KB)	Transmission Overhead
GenericApp	565	111	1
AME_GA_Mapped	1261	118	2.30
HomeAutomation	720	99+100	1
AME_HA_Mapped	1590	123+105	1.02

Table 6.2. Results on mapping efficiency.

code lines. The second column reports the number of significant calls to services provided by the underlying middleware: for the AME-based applications we considered `lookup()`, `registerobj()` and remote method invocations (e.g., `OnOffCB`) while for the original Z-stack applications we considered the invocations to Z-Stack API. Results show that source code is more compact and simple in case of SystemC applications because of its higher expressiveness with respect to C language. Clearly, this result would be meaningless without the possibility to translate SystemC code into native C code.

We assessed mapping efficiency by translating AME-based applications into C-language applications for the Texas Instruments' platform. `AME_GA_Mapped` has been generated from `AME_GenericApp`; `AME_HA_Mapped` has been generated from `AME_HomeAutomation`. Automatically-generated applications have been compared with the original Texas Instruments' examples and results are shown in Table 6.2. Considered metrics are the number of source code lines (first column), binary code size (second column), and transmission overhead with respect to the original Z-stack applications (third column). For the light-control example, binary code size has been reported for both light and switch components, respectively.

Binary code size is an important metric since sensor nodes usually have limited memory resources. Results show that the translation always increases the number of code lines and the binary code size but the limit of 128 KB is still satisfied.

Transmission overhead reveals the impact of translation on wireless communications. There is a significant difference of the transmission overhead between non-profile-enabled (2.30) and profile-enabled (1.02) applications. Without using Profiles, the emulation of the object-oriented programming paradigm requires more data transfers while profile-based applications already use part of these data transfers. In fact, the `AME_HA_Mapped` application increases the transmission overhead only for the use of the `lookup()` and `registerobj()` functions.

6.2 Mapping onto TeenyLime MW

TeenyLime is a middleware simplifying the development of WSN applications, and encompassing the peculiarities of sense-and-react scenarios. The foundation for TeenyLime is the notion of tuple space, a repository of elementary sequences of typed fields, called tuples. This notion is revisited in an original way by TeenyLime by considering the WSN requirements of dynamicity, resource consumption, and reliability in the programming model, and by satisfying them concretely through an efficient middleware implementation.

TeenyLime is written in nesC on top of TinyOS O.S.. nesC is an extension to the C programming language designed to embody the structuring concepts and execution model of TinyOS. TinyOS is an event-driven operating system designed for sensor network nodes that have very limited resources. A dedicated nesC interface (illustrated in Figure 6.9) is used (in the TinyOS sense) by the application to access the transiently shared tuple space composed of the local tuple space and that of the one-hop neighbors. Each nesC command requires a target, a specification of the tuple space repositories in the federation over which the operation should execute.

```

interface TupleSpace {

    // Standard operations
    command TLOpId_t out(bool reliable, TLTarget_t t, tuple *t);
    command TLOpId_t rd(bool reliable, TLTarget_t t, tuple *templ);
    command TLOpId_t in(bool reliable, TLTarget_t t, tuple *templ);

    // Reliable group operations
    command TLOpId_t rdg(bool reliable, TLTarget_t t, tuple *templ);
    command TLOpId_t ing(bool reliable, TLTarget_t t, tuple *templ);

    // Managing reactions
    command TLOpId_t addReaction(bool reliable, TLTarget_t t, tuple *templ);
    command TLOpId_t removeReaction(TLOpId_t operationId);

    // Request to reify a capability tuple
    event result_t reifyCapabilityTuple(tuple* ct);

    // Returning tuples
    event result_t tupleReady(TLOpId_t operationId, tuple *t, uint8_t n);
}

```

Fig. 6.9. TeenyLIME API.

The APIs `out()`, `rd()`, `in()` shown in Figure 6.9 representing the APIs `write()`, `read()`, `take()` typically provided by the Tuplespace programming paradigm as described in Section 3.2.2.

6.2.1 TeenyLime application

Figure 6.10 shows the template of a typical TeenyLime application implemented by using the following interfaces:

- `Tuplespace` has been described in Section 6.2; it allows to access TeenyLime's shared tuple space.
- `TeenyLIMESystem` provides the middleware with a neighbor tuple for the local host.
- `Timer` interface is used to trigger a periodic operation (e.g., reading values from a sensor or to implement a general operation).

```

includes TupleSpace;
module Template_Application {
  uses {
    interface TupleSpace;
    interface TeenyLIMESystem;
    interface Timer;
    ...
  }
  provides interface StdControl;
}
implementation {
  command result_t StdControl.start() {
    return call SensingTimer.start (TIMER_REPEAT, SENSING_TIMER);
  }
  event result_t StdControl.fired() {
    return call StdControl.run();
  }
  command result_t StdControl.run() {
    ...
  }
}

```

Fig. 6.10. Template of a typical TeenyLIME application.

The actual processing in the `Template_Application` module is fairly simple, as illustrated in the following fragment of code. `start()` command executes the `start()` command provided by the `Timer` interface in order to specify the type of timer (REPEAT or ONE SHOT) and the interval at which the timer will expire. Moreover the `Template_Application` module receives `fired()` event when the timer has expired and executes the `run` command (to create a tuple describing an information, to write it in the tuplespace by means of the `out` command, and so on).

6.2.2 Methodology

The *Mapping* process onto TeenyLime MW assumes that the programming paradigm remains the same in the transfer from abstract middleware (AME) to actual middleware; therefore this mapping is easier with respect to the mapping onto ZigBee described in Section 6.1.4, where calls to AME have to be replaced with direct calls to system SW (e.g., z-stack).

AME applications are written in SystemC (i.e., C++), while TeenyLime applications are written in nesC language. Therefore, the mapping process shown in Figure 6.11 consists of three key concepts in order to convert SystemC to nesC language:

1. During the first phase, the AME application is automatically translated into an intermediate format.
2. In the second step, the intermediate representation is manipulated to identify calls to Tuplespace services (`write()`, `read()`, `take()`) which must be translated with the corresponding TeenyLime APIs (`out()`, `rd()`, `in()`).
3. Finally, the intermediate representation is automatically converted into a nesC template application reported in Figure 6.10 which can be compiled and downloaded on the target platform.

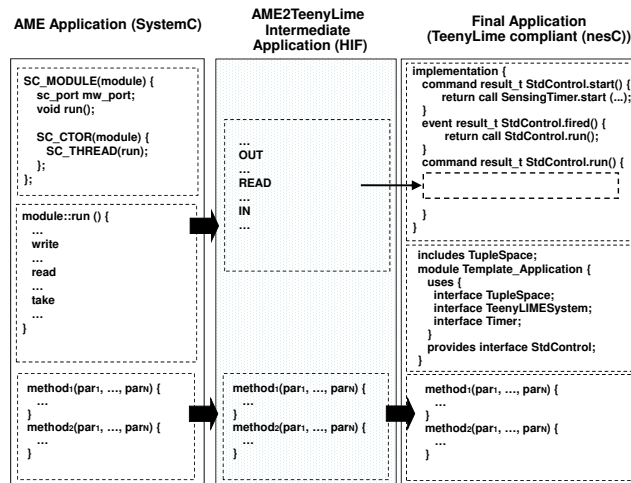


Fig. 6.11. AME to TeenyLIME mapping process.

The conversion is performed with the support of automatic tool implemented by using HIF-Suite.

Application to a heterogenous NES

The Middleware-centric desing methodology described in this work has been used to model a real NES, called Angel platform [82].

The Angel platform includes WSN to constitute the infrastructure needed to create a set of new services for life and health of the citizen. The architecture considered in the Angel platform consists of three main actors, i.e., the wireless ad-hoc network, the gateway and the remote agents. The ad-hoc wireless network consisting of nodes with different capabilities ranging from simple transmission nodes to full wireless sensor nodes; they communicate using short-range radio links. The ad-hoc wireless network accesses traditional communication networks through a dedicated node called Gateway. This node is responsible for configuring and querying the WSN, gathering data, processing them in collaboration with remote service centres and providing services to users. The Gateway can be embedded into mobile handsets or residential network appliances.

The gateway is used for the interconnection of a wireless sensor network with remote agents through a geographical network, to collect, aggregate and eventually pre-process data received by the WSN. It is also responsible for keeping personal information (in case of a mobile phone) and exact space position (in case of a fixed gateway). Personal information are used for service authentication and for user-driven services (e.g., to recover a personal profile or health disease). The exact space position can be used to locate sensor nodes.

Remote agents can be either collector of information or data/service providers. They are responsible for all those aspects of the distributed application which cannot be implemented on the wireless sensor nodes or on the gateway (e.g., large databases, computational intensive elaboration, need for human supervision). An example of remote agent is the remote destination of an alarm sent by the gateway after the wireless sensor network has detected a particular event. The Service Centre is also used to remotely configure and monitor the wireless sensor network, through the gateway.

The Middleware-centric design flow consists of several stages of increasing refinement level which produce different co-simulation Views.

Figure 7.1 shows View 1 of the proposed modelling flow for the whole Angel platform. The Gateway, the Remote Service and Sensor Nodes are modelled by SystemC to simplify the re-using of actual software; the whole Angel application

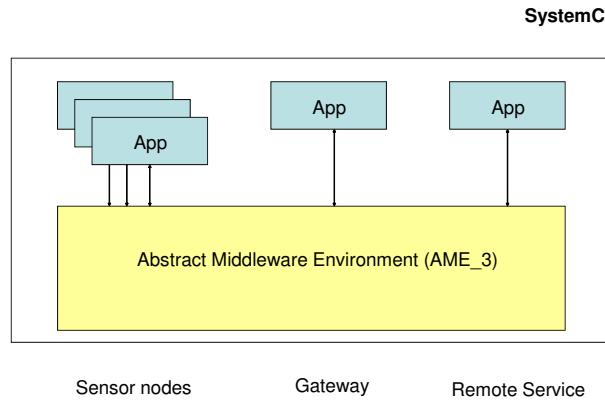


Fig. 7.1. View 1 of the modelling flow: the whole application is a set of interacting object-oriented modules.

is developed as a single distributed application in which communications between nodes are simulated through SystemC primitives; a possible approach could consist in creating a module for each network node, i.e., WSN nodes, the Gateway and the Remote Service. the application components on the different nodes communicate each other through an interoperable set of protocols provided by an underlying software layer named middleware; for example, in a temperature-monitoring application the piece of code which requests temperature samples, averages them and tests for dangerous conditions belongs to the Application while services to send queries and alarms belong to the middleware. To reflect this approach in the modelling flow, each node is partitioned into Application code and Abstract middleware; the former is the model of the actual application and it accesses lower services through the middleware API. The Abstract Middleware is similar to the middleware actually present on the final nodes in the fact that they provide the same API to the application; in this way, the models of the application components can be seamlessly ported to the actual platform; in other words, application developer can write modules and simulate them concurrently with the design of the actual HW/SW platform provided that application API has been previously defined. The Abstract Middleware is different from the actual middleware in the fact that it also reproduces the behaviour of HW and SW components of the actual platform. View 1 uses the Abstract Middleware Environment Level 3 (AME.3) in which communications are implemented through SystemC primitives.

Different types of middleware can be used; AME provides four programming paradigms: object-oriented, message-oriented, database and tuplespace. To simplify application development and the mapping of Abstract Middleware onto Actual Middleware we decided to use an object-oriented paradigm in the Abstract Middleware Environment; in fact several modern distributed applications consist

of distributed objects connected together by middleware such as CORBA, Web Services and Java RMI. Also, ZigBee applications follow a kind of object-oriented paradigm. For these reasons, in View 1 the application is designed as a set of classes interacting together.

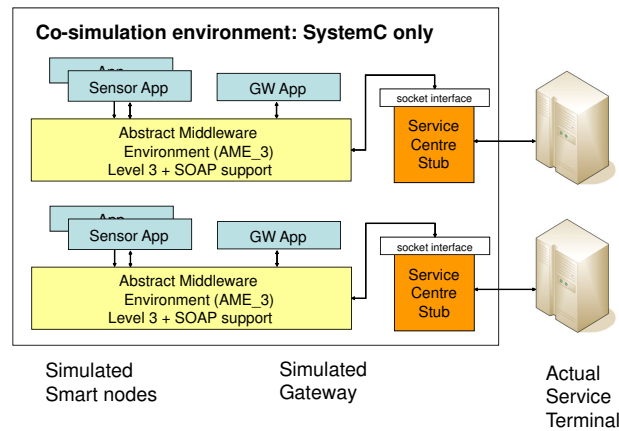


Fig. 7.2. View 2 of the modelling workflow: the Service Centre Stub allows the interaction with the actual Remote Service.

An important extension of View 1 is represented by the View 2 depicted in Figure 7.2. In this View the model of the application running on the Remote Service has been replaced by its actual implementation executed by a stand-alone server. Actual communications supported by the actual Remote Service application are re-directed to the SystemC simulator through the Service Centre Stub. This View can be used when the development of the Remote Service is almost completed and its complexity (e.g., the presence of a database server or of a web server) cannot be efficiently supported by SystemC; furthermore, this View can also be used to validate the Remote Service and to evaluate its impact on the Gateway. As depicted in the Figure, different instances of the pair WSN/Gateway can be created to reproduce complex scenarios.

In this View the Abstract Middleware is partially extended to be compliant with the middleware running on the Remote Service; specifically, the chosen object-oriented middleware is Web Services with the SOAP protocol [16].

The Service Centre Stub provides a standard BSD Socket API which is used by the abstract middleware of the Gateway to implement SOAP services.

Figure 7.2 also reports a minimal architecture of the Actual Service Terminal which consists of a standard PC executing a Java application and interacting with the Gateway through Web Services

Figure 7.3 shows the third View. Also in this case SystemC is the only modelling tool. The Abstract Middleware Environment Level 3 (AME.3) has been

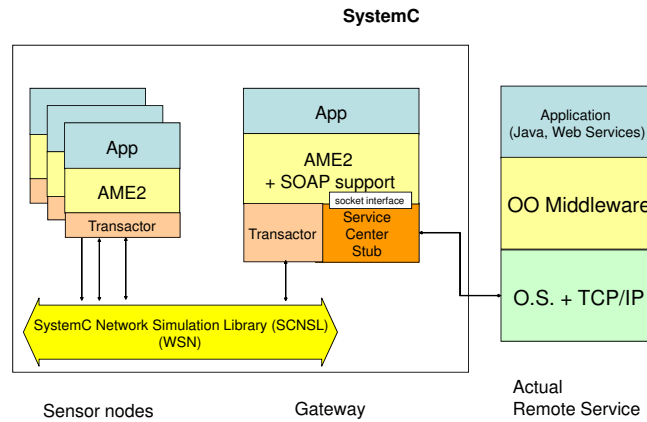


Fig. 7.3. View 3 of the modelling flow: WSN is modelled at packet level.

replaced with AME Level 2 (AME_2) in which communications are modelled in a more realistic way by a SystemC Network Simulation Library which reproduces packet transfers. This View allows evaluating the effect of design choices on communication performance (e.g., latency and throughput). It is worth noting that the interface between Application and Middleware remains unchanged with respect to the previous View; furthermore, since AME_2 will be used also in the following views the application code will not change in the all modelling flow. Application code and abstract middleware are pure software components and even though they are described by SystemC, there is no timing information inside them. Instead, the SystemC Network Simulation Library reproduces network behaviour with timing information (e.g., about propagation delay). To allow the interaction between un-timed and timed components an additional module named SCNSL-AME-Transactor (described in Section 4.2.3) is required as shown in Figure 7.2.

In View 4 (Figure 7.4) the approach of View 3 is extended also to the geographical network. Even though data to/from the Remote Service refer to its actual implementation, network performance can be evaluated through the SystemC Network Simulation Library. In this View other two transactors are required to interface the middleware and the Service Centre Stub to the SystemC Network Simulation Library.

In View 5 (Figure 7.5) the model of the Gateway is refined with a cycle-accurate emulation of the application software, the actual middleware and the operating system. The use of a CPU emulator increases the simulation accuracy for the software and it allows assessing the computation time, memory requirements, and power consumption as a function of SW complexity. We use a modified version QEmu [81] which is an open source emulation software to execute applications and operating system on a host platform; the advantage of QEmu is that the Gateway operating system can be fully emulated since most of the required HW components are

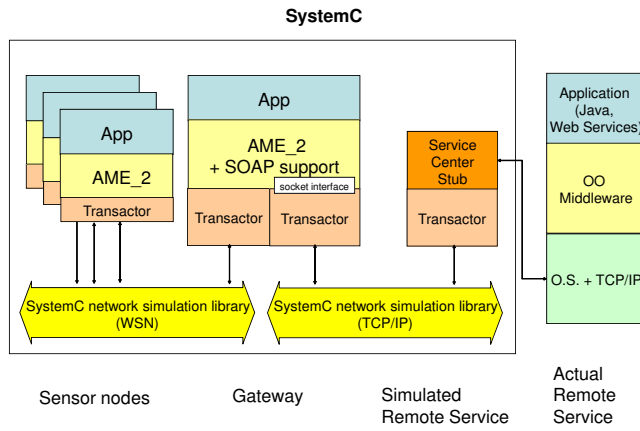


Fig. 7.4. View 4 of the modelling flow: both the WSN and the traditional network are modelled at packet level.

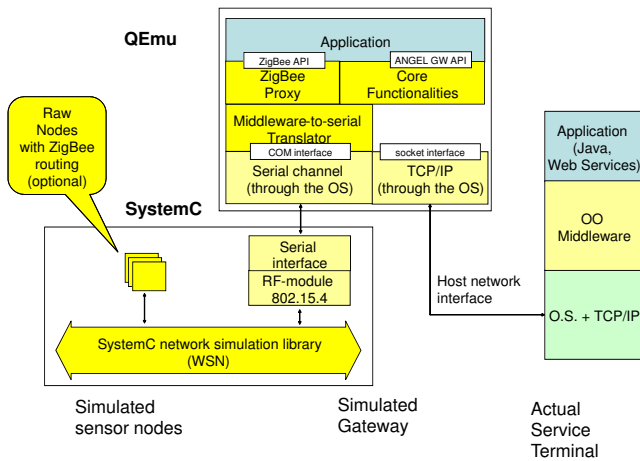


Fig. 7.5. View 5 of the modelling flow: cycle-accurate emulation of the Gateway.

mapped onto the host; for what concerns specific HW components, we modified QEmu, as described in Section 4.1.6, to interact with SystemC models. On the actual Gateway the ZigBee stack is executed by a different CPU which communicates with the main CPU through a serial channel; we decided to model the functionalities of this slave CPU through a SystemC component; for this reason, a new SystemC module has been added to implement a Serial Adapter to interface

the serial interface provided by QEmu and the 802.15.4 module to interface with the WSN.

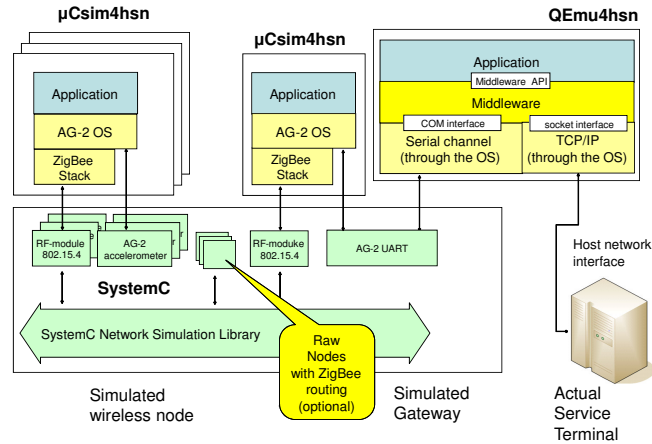


Fig. 7.6. View 6 of the modelling flow: cycle-accurate simulation of the Sensor Node and of the Gateway.

Figure 7.5 represents View 6 in which the model of the sensor node has been refined assuming that it is based on the Texas Instruments CC2430 System-on-Chip; we have introduced an instruction set simulator (ISS), named CSim, which executes the application software, the Aquisgrain OS, and the ZigBee stack contained in the official Aquisgrain software distribution. The ISS has been modified as described in Section 4.1.5, to interact with SystemC models representing the network interface (802.15.4 RF-module) and other HW components (i.e., the Aquisgrain accelerometer). The ISS version able to interact with SystemC is called uCsim4hsn. An instance of uCsim4hsn is also used to model the ZigBee interface of the Gateway; in this case a SystemC model of the Aquisgrain serial interface is used to link the Gateway CPU with the ZigBee slave CPU.

Conclusion

This thesis presented a middleware-centric design methodology for modelling and simulating of complex distributed applications based on a network of heterogeneous networked embedded systems. Based on the assumption that the presence of a middleware software simplifies the design of distributed applications, an Abstract Middleware Environment has been developed, named AME, providing services belonging to the common programming paradigms: tuplespace, publish-subscribe (MOM), object-oriented (OOM) and database. AME has been developed by using the SystemC system language.

The designer can thus rapidly develop a model of the application accordingly to the preferred paradigm. A proxy-based mechanism has been proposed to allow the designer to smoothly move across different programming paradigms in order to validate and simulate the NES applications. The translation methodology has been presented in order to implement this feature. It allows to automatically change the programming paradigm without re-writing the application code. This methodology can be used to de-couple the choice of the programming paradigm in the development phase from the use of a specific actual middleware; in fact, the choice of the programming paradigm during the development phase depends on the application designer skills and on the type/nature of the application; in the other hand, during the implementation phase, the choice depends on the actual platform chosen for the deployment.

Moreover, during the NES application modelling and simulation the model of the platform below the application can be detailed through three design steps: at the highest level (called AME_3) modules communicate through abstract point-to-point primitives; at the next level (called AME_2) *System/Network partitioning* is applied to the model; modules are mapped onto network nodes and communications between nodes are provided by AME_2 services through a network simulator. at the final level (called AME_1) *HW/SW partitioning* is applied to each node to map functionalities to HW and SW components.

The simulation of a networked embedded systems mixing sensors, gateway and service centers has shown the effectiveness of the proposed solution. Experimental results show that the efficiency of the programming paradigm is preserved across translation and therefore it can be evaluated during development as part of the design-space exploration.

Finally, after application development and design-space exploration, application code can be deployed over the actual NES by mapping AME calls to actual middleware calls. Two reference examples have been presented: the description of the automatic mapping of AME applications over a target NES platform running Z-Stack middleware (ZigBee) and the *Mapping* process onto a Tuplespace programming paradigm (named TeenyLime). In the first case, experimental results shown that this approach significantly simplifies the coding process: the resulting applications have also been compared with native and functionally equivalent implementations. The increase of binary code size is small (about 4%) since AME services are translated into services already provided by the ZigBee library, and even the transmission overhead is negligible whenever the mechanism of ZigBee Profiles can be exploited.

Concerning the future works, an aspect that should be investigated concerns the possibility to extend the programming paradigms supported by the Abstract Middleware Environment (e.g., component-based middleware). Moreover, from the practical point of view, it could be interesting to support several network simulators during the refinement process; therefore, it's necessary to extend the AME-transactor feature in order to be able to involve other network simulators. Finally, a mapping solution for each programming solution could be investigated; nowadays, ZigBee and TeenyLime are supported.

References

Published contributions

1. F.Fummi, M.Loghi, **G.Perbellini**, M.Poncino, *ISS-Centric Modular HW/SW Co-Simulation*, "ACM Great Lakes Symposium on VLSI (GLSVLSI)", Philadelphia, PA, 30 April - 02 May, 2006.
2. F. Fummi, **G.Perbellini**, R. Pietrangeli, D. Quaglia, *A Middleware-Centric Design Flow for Networked Embedded Systems*, "IEEE Design Automation and Test in Europe Conference (DATE)", Acropolis, Nice, France, 16-20 April, 2007.
3. F. Fummi, **G.Perbellini**, D. Quaglia, S. Vinco, *AME: an Abstract Middleware Environment for validating Networked Embedded Systems Applications*, "IEEE International High Level Design Validation and Test Workshop (HLDVT07)", Irvine, November, 2007.
4. F. Fummi, M. Loghi, **G.Perbellini**, M. Poncino, *SystemC co-simulation for core-based embedded systems*, "An International Journal on Design Automation for Embedded Systems", Springer-Verlag, Volume 11, Numbers 2-3 September, 2007.
5. A. Bragagnini, F. Fummi, A. Huebner, **G.Perbellini**, D. Quaglia, *Co-simulation framework for the Angel platform*, "IEEE International Conference on Electronics, Circuits and Systems (ICECS)", Marocco, December, 2007.
6. E. Alessio, A. Bragagnini, **G.Perbellini**, D. Quaglia, *Gateway and Middleware Design: trusted WSN-TLC network communication and enhanced WSN management*, "IEEE International Conference on Electronics, Circuits and Systems (ICECS)", Marocco, December, 2007.
7. A. Acquaviva, F. Fummi, **G.Perbellini**, D. Quaglia, *An Energy-Aware Co-Simulation Framework for the Design of Wireless Sensor Networks*,

- "ACM Great Lakes Symposium on VLSI (GLSVLSI)", Orlando, Florida , May 4-6, 2008 , 2008 , pp. 375-378.
8. S. Cordibella, F. Fummi, **G.Perbellini**, D. Quaglia, *A HW/SW Co-Simulation Framework for the Verification of Multi-CPU systems*, "IEEE International High Level Design Validation and Test Workshop (HLDVT)", Nevada, November, 2008.
 9. F. Fummi, **G.Perbellini**, N. Roncolato, *Networked Embedded System Applications Design Driven by an Abstract Middleware Environment*, "IEEE Design Automation and Test in Europe Conference (DATE)", Acropolis, Nice, France, 20-24 April, 2009.
 10. A. Acquaviva, F. Fummi, **G.Perbellini**, D. Quaglia, *Flexible Energy-Aware Simulation of Heterogeneous Wireless Sensor Networks*, "IEEE Design Automation and Test in Europe Conference (DATE)", Acropolis, Nice, France, 20-24 April, 2009.
 11. F. Fummi, **G.Perbellini**, D. Quaglia, S. Vinco, *A SystemC-centric Approach for Generation and Simulation of WSN Applications Targeted to ZigBee*, [Submitted for review]
 12. F. Fummi, **G.Perbellini**, D. Quaglia, R. Trenti, *Network Simulation Refinement for Middleware-centric Networked Embedded Systems application*, [Submitted for review]
 13. F. Fummi, **G.Perbellini**, D. Quaglia, S. Saggin, S. Vinco, *Mixing Modeled and Actual Hardware Devices to Co-simulate a Complex Embedded Platform*, [Submitted for review]

Bibliography

14. Martin G., Lennard C., "Improving embedded software design and integration in SOCs", *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC 2000)*, pp. 101-108, May, 2000.
15. Twan Basten, Marc Geilen, Harmke de Groot, "Ambient Intelligence: Impact on Embedded System Design", *Kluwer Academic Publishers*, 2003.
16. P. Volgyesi, A. Ledeczki, "Component-Based Development of Networked Embedded Applications", *In Proc. of Euromicro conference*, 2002, pp. 68-73.
17. Paolo Costa, Luca Mottola, Amy L. Murphy and Gian Pietro Picco. "Programming Wireless Sensor Networks with the TeenyLIME Middleware", *in Proceedings of the 8th ACM-IFIP-USENIX International Middleware Conference (Middleware 2007)*, Newport Beach (CA, USA), November 26-30, 2007.
18. E. Souto et al., "A message-oriented middleware for sensor networks", *in Proc. of the Workshop on Middleware for Pervasive and Ad-Hoc Computing*, 2004, Vol. 77, pp. 127-134.

19. Kay Rmer, "Programming Paradigms and Middleware for Sensor Networks", <http://citeseer.ist.psu.edu/666689.html>, 2004.
20. V. Subramonian et al., "Middleware Specialization for Memory-Constrained Networked Embedded Systems", *In Proc. of IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004, pp. 306-313.
21. Schantz R., Schmidt D., "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," *Encyclopedia of Software Engineering*, Wiley & Sons, 2001.
22. Schmidt D., and S. Huston. "C++ Network Programming: Resolving Complexity with ACE and Patterns". *Addison-Wesley*, 2002.
23. G. Bollella, B. Brosgol, P. Dribble, and et. al. "The RealTime Specification for Java. The Java Series". *Addison-Wesley*, 2000.
24. Sun Microsystems, Inc., "Java Remote Method Invocation Specification", *Sun Microsystems*, October 1998.
25. Object Management Group, "Common Object Request Broker Architecture: Core Specification". March 2004, *Version 3.0.3 - Editorial changes formal04-03-01*.
26. Object Management Group, "CORBA-Services: Common Object Service Specification". *OMG Technical Document Formal98-12-31*.
27. Object Management Group, "CORBA Component Model Joint Revised Submission". *OMG Document orbos99-07-01*.
28. Salem Hadim and Nader Mohamed, "Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks", *IEEE Distributed Systems Online*, 7(3), Mar. 2006.
29. J. Gehrke, S. Madden, "Query processing in sensor networks", *IEEE Pervasive Computing*, Vol. 3, No. 1, pp. 46-55, 2004.
30. S.R. Madden et al., "TinyDB: An Acquisitional Query Processing System for Sensor Networks", *ACM Trans. Database Systems*, Vol. 20, No. 1, pp. 122-173, 2005.
31. David Gelernter, "Generative communication in Linda", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80-112, 1985.
32. T. J. Lehman, Stephen W. McLaughry, and Peter Wyckoff, "T Spaces: The Next Wave." *In Proc. of the Int. Conference on System Sciences*, 1999.
33. C. Curino, M. Giani, M. Giorgetta, A. Giusti, A.L. Murphy, and G.P. Picco, "TinyLime: Bridging Mobile and Sensor Networks through Middleware," *In Proc. of the IEEE International Conference on Pervasive Computing and Communications (PerCom)*, Kauai Island, Hawaii, USA, pp. 61-72, March 2005.
34. C. Mascolo, L. Capra, W. Emmerich, "Mobile Computing Middleware (A Survey)." *In: Advanced Lectures on Networking*, (NETWORKING 2002), Pisa, Italy, 2002.
35. Michi Henning, "A New Approach to Object-Oriented Middleware", *IEEE Internet Computing*, Vol. 8, No. 1, pp. 66-75, 2004.
36. D. C. Schmidt et al., "TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems." *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

37. Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic, "A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems", *IEEE Transactions on Software Engineering*, Vol. 31, NO. 3, March 2005
38. D. Schmidt, "Model-driven engineering", *IEEE Computer*, 39(2), Feb 2006.
39. Woo Yeol Kim, R. Young Chul Kim, "Adapting Model Driven Architecture for Modeling Heterogeneous Embedded S/W Components", *International Conference on Hybrid Information Technology (ICHIT'06)*, Volume 2, pp. 705-711, Nov. 2006.
40. Balasubramanian K., Gokhale A., Karsai G., Sztipanovits J., Neema S., "Developing Applications Using Model-Driven Design Environments", *IEEE Computer*, Volume 39, N. 2, pp. 33-40, 2006.
41. Hasiotis T., Alyfantis G., Tsetsos V., Sekkas O., Hadjiefthymiades S., "Sensation: a middleware integration platform for pervasive applications in wireless sensor networks", *Proceedings of the Second European Workshop on Wireless Sensor Networks, 2005*, pp. 366-377, Jan., 2005.
42. Issarny Valerie, Caporuscio Mauro, Georgantas Nikolaos, "A Perspective on the Future of Middleware-based Software Engineering", *Future of Software Engineering, 2007. FOSE '07* pp. 244 - 258, 2007.
43. Donghee Kim, Chang-Eun Lee, Jun Hee Park, KyeongDeok Moon, Kyungshik Lim, "Scalable Message Translation Mechanism for the Environment of Heterogeneous Middleware", *IEEE Transactions on Consumer Electronics*, Vol. 53, No. 1, Feb. 2007.
44. M. Sgroi, Adam Wolisz, Alberto Sangiovanni-Vincentelli and Jan M. Rabaey, "A Service-Based Universal Application Interface for Ad-hoc Wireless Sensor Networks", whitepaper, U.C.Berkeley, 2004.
45. T. Grotker, S. Liao, G. Martin, S. Swan, "System Design with SystemC", *Springer*, 2002.
46. S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Extending the Transaction Level Modeling approach for fast communication architecture exploration", *In Proc. of the Design Automation Conference (DAC)*, pp. 113-118, 2004.
47. P. Levis et al., "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Application", *in Proc. of SENSYS*, 2003.
48. F.Fummi et al., "A Timing-Accurate Modeling and Simulation Environment for Networked Embedded Systems", *in Proc. of the IEEE Design Automation Conference (DAC)*, 2003, pp. 42-47.
49. J. Siegel, "Why Use the Model Driven Architecture to Design and Build Distributed Applications ?", *in Proc. of the Int. Conf. on Software Engineering*, 2005, pp. 37-37.
50. L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "Systemc cosimulation and emulation of multiprocessor SoC designs", *IEEE Computer*, 36(4):5359, Apr. 2003.
51. D. Bertozzi et al., "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip", *IEEE Trans. Parallel Distrib. Syst.*, 16(2):113129, Feb. 2005.
52. M. Conti and D. Moretti, "System level analysis of the Bluetooth standard", *In Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, volume 3, pages 118123, Mar. 2005.

53. S. McCanne and S. Floyd. "NS Network Simulator version 2". URL: <http://isi.edu/nsnam/ns/>.
54. S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. "GHT: A geographic hash table for datacentric storage". In *Proc. of the 1st ACM Int. Workshop on Wireless Sensor Networks and Applications*, 2002.
55. C.-Y. Wan, A. T. Campbell, and L. Krishnamurthy. "PSFQ: A reliable transport protocol for wireless sensor networks". In *Proc. of the 1st ACM Int. Workshop on Wireless Sensor Networks and Applications*, 2002.
56. S. Park, A. Savvides, and M. Srivastava. "Sensorsim: a simulation framework for sensor networks". In *Proc. of 3rd ACM Int. Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 104111, 2000.
57. I. Downard. "Simulating Sensor Networks in NS-2". *NRL/FR/552204-10073*, Naval Research Laboratory, Washington, D.C., May 2004.
58. H. L. F. Ye, J. Cheng, S. Lu, and L. Zhang. "A two-tier data dissemination model for large-scale wireless sensor networks". In *Proc. of the 1st ACM Int. Workshop on Wireless Sensor Networks and Applications*, 2002.
59. H. Weinberg, "Using the ADXL202 in Pedometer and Personal Navigation Applications", *Analog Devices Application Note*.
60. 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 Design and Automation Conf. (DAC)*, pages 4247, Jun. 2003.
61. F. Fummi, M. Loghi, S. Martini, M. Monguzzi, G. Perbellini, and M. Poncino. "Virtual hardware prototyping through timed hardware-software co-simulation". In *Proc. of the IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE)*, volume 2, pages 798803, Mar. 2005.
62. F. Fummi, S. Martini, G. Perbellini, M. Poncino, F. Ricciato, and M. Turolla. "Heterogeneous co-simulation of networked embedded systems". In *Proc. IEEE Design Automation and Test in Europe Conf. (DATE)*, pages 168173, Feb. 2004.
63. F. Fummi, D. Quaglia, F. Ricciato, and M. Turolla. "Modeling and simulation of mobile gateways interacting with wireless sensor networks". In *Proc. IEEE Design Automation and Test in Europe Conf. (DATE)*, Mar. 2006.
64. D. Dietterle, J. Ebert, G. Wagenknecht, R. Kraemer, "A wireless communication platform for long-term health monitoring", In *Proc. IEEE International Conference on Pervasive Computing and Communications Workshop*, Mar. 2006.
65. G. Nicolescu, S. Yoo, and A. A. Jerraya, "Mixed-level cosimulation for fine gradual refinement of communication in SoC design", In *Proc. IEEE Conf. on Design, Automation and Test in Europe (DATE)*, 2001, pp. 754759.
66. H. Posadas et al. "System-level performance analysis in SystemC". In *Proc. of IEEE Design Automation and Test in Europe and Exhibition*, pages 378383, Feb. 2004.
67. M. Bacivarov, S. Yoo, and A. Jerraya. "Timed HW/SW cosimulation using native execution of OS and application SW". In *Proc. of IEEE High-Level Design Validation and Test Workshop*, pages 5156, Oct. 2002.

68. M. Chung and C.-M. Kyung. "Enhancing performance of HW/SW cosimulation and coemulation by reducing communication overhead". *IEEE Transactions on Computers*, 55(2):125136, Feb. 2006.
69. Y. Nakamura et al. "A fast hardware/software co-verification method for system-on-chip by using a C/C++ simulator and FPGA emulator with shared register communication". In *Proc. of IEEE Design Automation Conference*, pages 299-304, Jun. 2004.
70. P. S. Magnusson et al. "Simics: A full system simulation platform". *IEEE Computer*, 35(2):5058, Feb. 2002.
71. L. Benini et al. "MPARM: exploring the multi-processor SoC design space with SystemC". *Journal of VLSI Signal Processing Systems*, 41(2):169182, 2005.
72. F. Fummi, M. Loghi, G. Perbellini, and M. Poncino. "ISS-centric modular HW/SW co-simulation". In *Proc. of ACM Great Lakes Symposium on VLSI (GLSVLSI)*, May 2006.
73. "SystemC Network Simulation Library (SCNSL)", <http://sourceforge.net/projects/scnsl/>.
74. LAN/MAN Standards Committee of the IEEE Computer Society, "IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs)", Sept., 2006.
75. ZigBee Standards Organization, "ZigBee 2006 Specification, Revision 13," URL: http://www.zigbee.org/en/spec_download/download_request.asp, 2006.
76. University of California at Berkeley, "TinyOS Operating System", URL: <http://www.tinyos.net>.
77. F. J. Villanueva, et al., "Lightweight Middleware for Seamless HW-SW Interoperability with Application to Wireless Sensor Networks", in *Proc. of the IEEE Design Automation and Test in Europe Conference (DATE)*, Acropolis, Nice, France, 16-20 April, 2007.
78. Texas Instruments Inc., "Z-Stack - ZigBee Protocol Stack", <http://focus.ti.com/docs/toolsw/folders/print/z-stack.html>.
79. G. Braun et al., "A universal technique for fast and flexible instruction-set architecture simulation", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 12, pp. 1625-1639, Dec. 2004.
80. M. Monton, A. Portero, M. Moreno, B. Martinez, and J. Carrabina. "Mixed SW/SystemC SoC Emulation Framework", *ISIE 2007: International Symposium on Industrial Electronics*, June 2007, pp. 2338-2341.
81. QEMU Emulator, <http://fabrice.bellard.free.fr/qemu>.
82. , European Commission, "Advanced Networked embedded platform as a Gateway to Enhance quality of Life - ANGEL, FP6-2005-IST-5-033506-STP, URL: <http://www.ist-angel-project.eu>, 2005.
83. A. Rubini and J. Corbet, "Linux Device Drivers, 2nd Edition". O'Reilly, June 2001.
84. EDALab - Networked Embedded Systems, "HIF Suite", www.edalab.it.