Networked Embedded System Applications Design Driven by an Abstract Middleware Environment *

F. Fummi, G. Perbellini, Niccolo' Roncolato

University of Verona - Department of Computer Science - Strada le Grazie, 37134, Verona, Italy
{franco.fummi,giovanni.perbellini}@univr.it

Abstract

The extreme heterogeneity of networked embedded platforms makes both design and reuse of applications really hard. These facts decrease portability. A middleware is the software layer that allows to abstract the actual characteristics of each embedded platform. Using a middleware decreases the difficulty in designing applications, but programming for different middlewares is still a barrier to portability. This paper presents a design methodology based on an abstract middleware environment that allows to abstract even the services provided. This is gained by allowing the designer to smoothly move across different design paradigms. As a proof, the paper shows how to mix and exchange applications between tuple-space and message-oriented based middleware environments.

1 Introduction

Typical pervasive computing technologies (such as telemedicine, manufacturing, crisis management) are part of of a broader class of networked embedded systems (NES) in which a large number of nodes are connected together and collaborate to perform a common task under a defined set of constraints. Such networked embedded systems cannot be realized without enabling middleware technologies. Middleware has emerged as an important architectural component in supporting NES applications. The role of middleware is to present a unified programming model to application writers and to mask out the problems of heterogeneity and distribution. Several NES middleware have been implemented in the past [1, 2], each one providing different programming paradigms (e.g., tuplespace, message-oriented, object-oriented, database, etc.). However, the diversity of properties provided by the state of the art makes the development of high quality middleware-based software systems complex: 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 [3]: (i) The selection of the middleware to be used should be based on engineering methods and on the system requirements; (ii) System design, specification and analysis must integrate properties manage at middleware level; (iii) System implementation should be built as much as possible on middleware tools; (iv) System validation needs to be performed integrating both middleware-related and application specific components.

To reduce implementation costs across families of different networked embedded systems, the implementation must also be re-usable across a variety of network topologies/infrastructures and HW/SW features.

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

Model-Driven Engineering (MDE) is a significant step towards a middleware-based software process [4]. The best known MDE initiative is the Model-Driven Architecture (MDA) from the Object Management Group (OMG). Using the MDA 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

PrismMW [5] 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

*Research activity partially supported by the FP6-2005-IST-5-033506 ANGEL European Project

978-3-9810801-5-5/DATE09 © 2009 EDAA

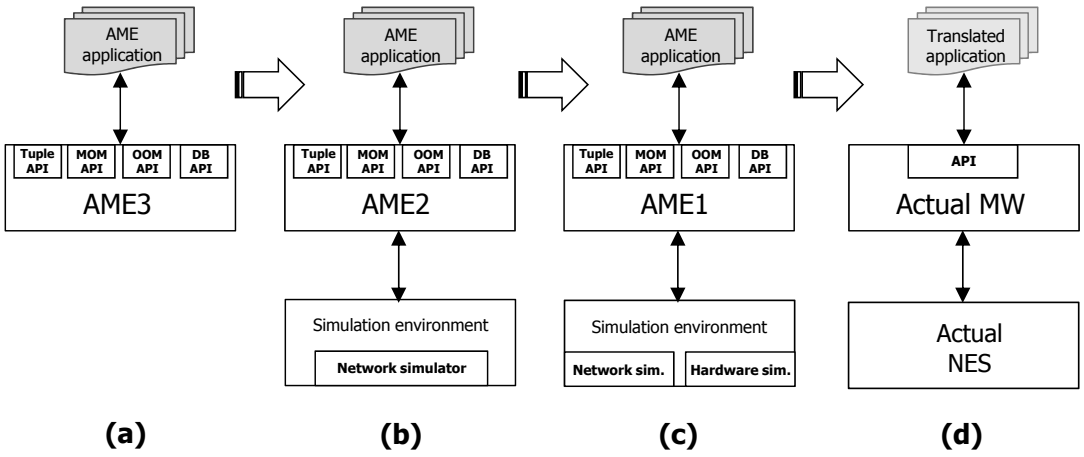

Figure 1. AME design flow.

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 [6] 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.

In this paper we present Abstract Middleware Environment (AME), a framework that abstracts common programming paradigms to design NES applications: tuplespace, publish-subscribe (MOM), object-oriented (OOM) and database. AME represents an enhanced and extended version of the simulation environment implemented in SystemC described in [7]. First, it allows to validate the functional behaviour of the NES applications. Moreover, AME helps the designer of NES applications to choose the programming paradigm for the development of the SW system, in order to satisfy system's functional and non-functional properties (e.g., synchronous or asynchronous communication, tight or loosely coupled interaction). For instance, an asynchronous communication requirement would not to 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. Finally, AME introduces the concept of the proxy-middleware making NES applications interoperable on different devices.

The remainder of this paper is organised as follows. Section 2 describes an overview of AME and the implementation of the proxy-middleware; Section 3 reports the translation between two of the programming paradigms provided by AME (Tuplespace and MOM). Finally, Section 4 presents the result of the translation mechanism provided by AME and applied to a real NES platform.

2 Abstract Middleware Environment

Abstract Middleware Environment 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 AME3), depicted in Figure 1.a, simulates and validates application functional requirements by using middleware-like services with different programming paradigms;
- During the second step (AME2), a simulated network infrastructure is involved in the whole framework, as shown in Figure 1.b. AME2 provides the same API of the previous step, even if opportunely modified to establish a communication with a network simulator (e.g., NS2 [8], SCNSL [9]).
- At the third step (called AME 1) 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 are provided by AME 1 services through HW/SW/network simulators (Figure 1.c).

It is worth noting that AME design flow allows to include step-by-step the NES model at different abstraction layers, while the application software remains un-changed. Indeed, the interface between applications and AME is well-defined. Thus, different development teams can work on both the application and the NES at the same time.

When the AME-based design application is completed, a mapping process (Figure 1.d) is implemented to deploy the application over the actual NES. If the HW/SW resources of the actual platform (e.g., a particular WSN) allow the presence of an actual middleware, then the application code is automatically translated to replace calls to AME services with calls to the actual middleware. The described design flow allows design space exploration to evaluate the performance of the same application by using different programming paradigms. This feature, called translation, gives the freedom to choose the programming paradigm preffered by the designer during the NES application design. The translation process is enabled by the introduction of a new module named proxy-MW (AME Proxy Middleware) w.r.t. the work described in [7].

2.1 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 2 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.

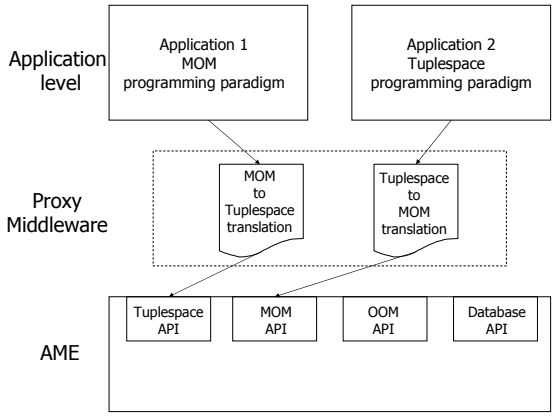

Figure 2. Proxy middleware.

Let us suppose to translate a MOM application into a Tuplespace application (the translation methodology will be described in Section 3). 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 as described in [7], easing the translation mechanism. This feature also guarantees a reduction of the translated applications in term of 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.

3 Translation

In this Section the translation between Tuplespace and Message-Oriented-Middleware program paradigms will be described (the other translation mechanisms have been presented in [7]). First of all, the two programming paradigms used are described. The Tuplespace model uses a globally-shared, associatively-addressed memory space, called Tuplespace. Inter-node communications take place by writing and reading tuples into the space. The services provided by this programming paradigm are:

- `read (template)`: to read a tuple from tuplespace corresponding to a given template; it can be both blocking and non-blocking;
- `take (template)`: to read and remove a tuple from tuplespace; it can be both blocking and nonblocking;
- `write (tuple)`: to add a tuple to tuplespace.

Message-oriented-Middleware uses a Publish/Subscribe paradigm. It is an asynchronous messaging paradigm in which publishers post messages to an intermediary broker and subscribers register subscriptions with the same broker. In a topic based system, messages are published to "topics" or named logical channels which are hosted by a broker. Subscribers will receive all messages published to the topics to which they subscribe and all subscribers to the same 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 depicts the communication flow between two applications (Node A and Node B) implemented by using the AME MOM programming paradigm.

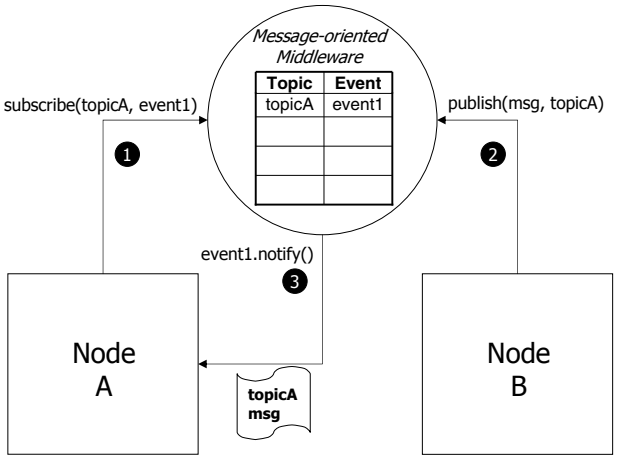

Figure 3. MOM general schema.

3.1 Tuplespace to MOM translation

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. 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 4.

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 draw 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 4.

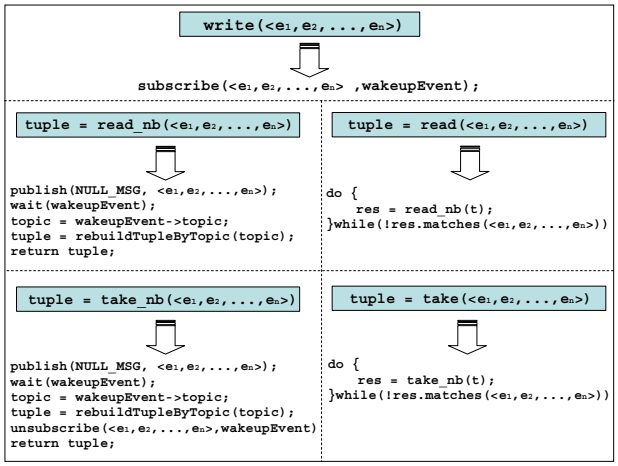

Figure 4. Tuplespace to MOM.

3.2 MOM to Tuplespace translation

MOM adopts a different communication paradigms w.r.t. 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:

$$< data, topic >$$

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 w.r.t. 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. 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. 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.

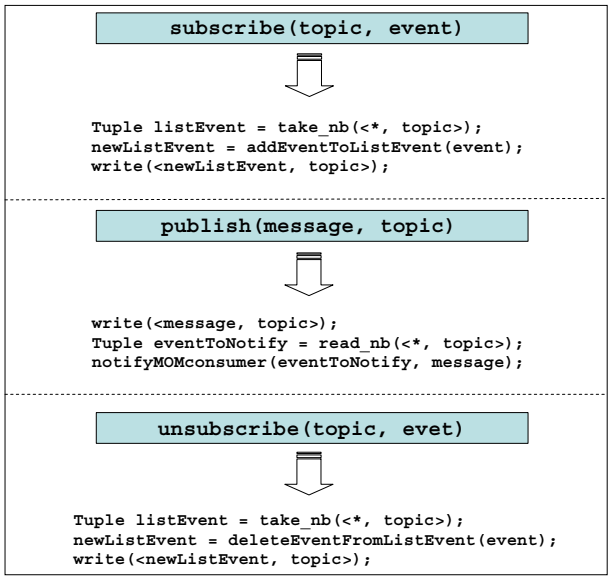

Figure 5. MOM to Tuplespace.

4 Experimental analysis

The first step (AME3) of the proposed middleware-centric design flow has been applied to a home care system scenario depicted in Figure 6. The objective is the evaluation of the translation mechanism described in 3, pointing up the advantages of the proxy-MW solution.

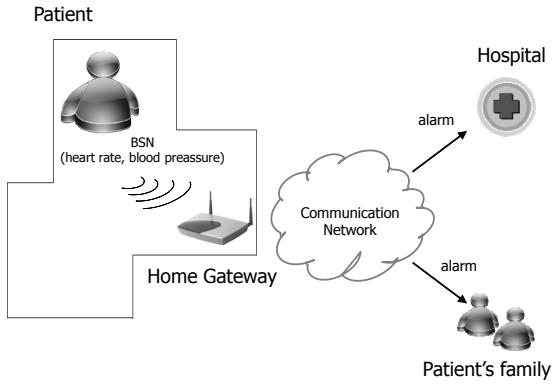

Figure 6. Home care system.

In the home care system scenario the patient wears a specific Body Sensors Network (BSN) to measure the vital-sign parameters (heart rate or blood pressure). The captured data are transmitted to the Home Gateway (HG) in order to continuously monitor and record the patient's condition. If the vital parameters overcome a fixed threshold, an alarm is sent to the Service Centre (SC), such as patient's family or the hospital so that they can rescue the patient.

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. Then, each original model has been translated applying the proxy-MW solution previously described. All translations between the different programming paradigms have been executed (except the translation starting from DB services to MOM and OOM).

Table 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.

	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

Table 1. Simulation performance results of the programming paradigms translations.

For example, the proxy-MW used to translate a tuplespace application into a MOM application includes the pseudo-code described in Section 3.1 equal to 220 code lines.

Figure 7 shows the performance of AME proposed in this paper with respect to [7], 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 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 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 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.).

5 Conclusions

The paper presented a design environment for modelling and simulating networked embedded systems applications. The environment is based on an abstract middleware (AME) providing services belonging to different programming paradigms. A proxy-based mechanism has been

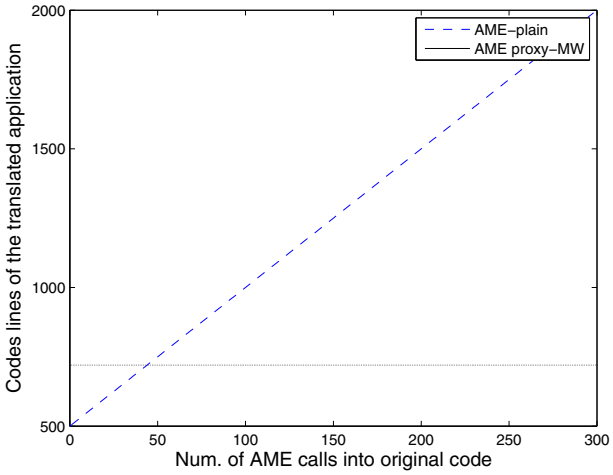

Figure 7. AME proxy-MW vs. AME plain.

proposed to allow the designer to smoothly move across different design paradigms in order to validate and simulate the NES applications. The paper exemplified how to translate applications based on Tuplespace to Message-oriented and viceversa. The proxy-based abstract midldeware environment allows the designer to mix applications written in tuplespace, object-oriented, message-oriented and database, thus allowing an easy application portability. The simulation of a networked embedded systems mixing sensors, gateway and service centres has shown the effectiveness of the proposed solution.

References

[1] 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.

[2] E. Souto, G. Guimaraes, G. Vasconcelos, "A message-oriented middleware for sensor networks", *in Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-hoc Computing*, Vol. 77, 2004.

[3] 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, N0. 3, March 2005

[4] D. Schmidt, "Model-driven engineering", *IEEE Computer*, 39(2), Feb 2006.

[5] 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.

[6] 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.

[7] Fummi, F.; Perbellini, G.; Quaglia, D.; Vinco, S.; "AME: an abstract middleware environment for validating networked embedded systems applications", *IEEE International High Level Design Validation and Test Workshop, 2007. HLVDT 2007*, pp. 187 - 194, 7-9 Nov. 2007.

[8] "The Network Simulator - NS2", *http://www.isi.edu/nsnam/ns/*.

[9] "SystemC Network Simulation Library (SCNSL)", *http://sourceforge.net/projects/scnsl/*.