Rialto: a Bridge between Description and Implementation of Control Algorithms for Wireless Sensor Networks

Alvise Bonivento* Luca P. Carloni Alberto L. Sangiovanni-Vincentelli

ABSTRACT

Rialto is a design framework that allows separating the description of a control application for wireless sensor networks from its physical network implementation. The methodology supported by Rialto consists of two steps:

1. An application is described in a Rialto Model in terms of logical components queries and commands.
2. The description is translated into an internal format called RialtoNet that is used to explore all the possible sequence of queries and commands that the application may lead to. The RialtoNet is executed and a set of constraints on the communication and sensing infrastructure is generated.

The semantics of RialtoNet is based on a MoC that takes inspiration from Kahn Process Networks, but blocking rules are conveniently modified to exploit the domain specificity.

Our approach offers a clear interface to the application designer as Rialto automatically bridges the gap between application and implementation. Hence, Rialto facilitates the adoption of wireless sensor networks technology in application domains, such as industrial control, where the application designer is not a communication engineer.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Distributed Networks, Wireless Communications*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages, Methodologies, Tools*

General Terms

Design, Languages

*A. Bonivento and A. Sangiovanni-Vincentelli are with the EECS Department of U.C. Berkeley, Berkeley, CA 94720; alvise,alberto@eecs.berkeley.edu; www-cad.eecs.berkeley.edu/∼alberto; L. Carloni is with the Department of Computer Science of Columbia University in the City of New York, NY 10027 luca@cs.columbia.edu; www.cs.columbia.edu/∼luca.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.
Copyright 2005 ACM 1-59593-091-4/05/0009 ...$5.00.

Keywords

Sensor Networks, Design Automation, Design Flow

1. INTRODUCTION AND BACKGROUND

Although wireless technology experienced great advancements in the last years, several industrial domains have been barely touched by it. A serious impediment is the complexity of mapping an application on a wireless network for the lack of appropriate abstraction levels that could insulate the application engineer from the drudgery of the implementation architecture. This paper addresses this very problem offering a methodology, an environment and supporting tools to map an application on a wireless sensor network.

Consider the case of automotive manufacturing plants. In these plants a huge number of sensors is deployed to monitor and control the state of robots in the production line. These sensor networks are usually wired and, as a consequence, the plant is characterized by a large amount of cables. Besides high maintenance and deployment costs, this solution also has a reverse impact on flexibility (the addition of new sensors and reconfiguration implies stopping the production line) and safety (human operators may trip on cables). The behavior of these networks is decided by a cyclic control routine implemented in a Controller that is usually placed inside the plant. The control routine evaluates the sensed data and takes decisions on the next action to take to preserve the proper working conditions. The routine is typically characterized by a high level of computational complexity, but a low number of possible decisions (i.e. move the robot up, or down, or switch it off).

The software for these applications is usually written by process or mechanical engineers that are expert in process control technology, but know little of the communication and sensing infrastructure that has to be deployed to support these algorithms. On the other side, the communication infrastructure is designed by communication engineers that know little about process control technology. Moreover, the adoption of wireless technology further complicates the design of these networks. Being able to satisfy high requirements on communication performance over an extremely unreliable communication channel is a difficult task. Consequently, the gap between the control algorithm designers and the network designers will inevitably increase and this phenomenon might delay the adoption of wireless sensor networks technology within manufacturing plants. The Rialto project is specifically aimed at bridging this gap.

Several tools to support the design of WSNs are available. The most common design methodology for WSNs

starts with the description of the protocol specifications using the NesC/TinyOS stack [6]. The NesC/TinyOS platform was developed at U.C. Berkeley and it leverages a "method call" model of computation. It was designed to describe component-based architectures using a simple event-based concurrency model. This platform has then been enriched with a simulation environment called TOSSIM [7]. Its success is also related to the wide spreading of the hardware platforms of the Mica family [5].

Alternatively, protocol solutions are simulated using environment such as OMNET++ [8] or VisualSense [9] and then implemented in NesC/TinyOS. Omnet++ is a discrete event simulator developed by Andras Varga at the Technical University of Budapest. Although it is not target specifically for WSNs, Omnet++ is widely used within the communication community for protocol simulations. Visualsense is a modeling framework for WSNs developed as part of the Ptolemy project at U.C.Berkeley [10]. It is an extension of a discrete-event model with the extra capability of describing properties of the wireless connectivity. Visualsense is a powerful tool to model and evaluate protocol solutions under different scenarios.

An attempt of raising the level of abstraction was made by Yu et al. in [14], where a classification for node communication mechanisms was introduced to allow for a higher level description of the network algorithms. In [15], a design methodology was presented. That methodology is based on a bottom-up part for the description of network algorithms, a top-down part to describe the application, and a mapping process to define the code that must be deployed on the nodes. Although we share the idea of the coexistence of a top-down and bottom-up approach, we believe that these approaches are too network oriented and not enough application oriented. Our approach enphasizes the control based nature of WSN applications and offers a clear semantic and set of primitives to interpret timing issues at a very high level, hence providing a clear level of abstraction for the application designer. In particular, the design flow that we envision is summarized in Fig. 1:

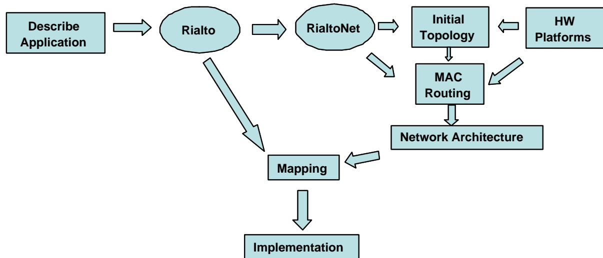

Figure 1: Design Flow for WSN

The application engineer describes the control application independently from the particular communication infrastructure or hardware platform with the Rialto Model.

Rialto capture these specifications in a formal way and perform a state space exploration to analyze all the possible scenarios that the application may lead to. As a result of this exploration, produce a set of constraints that the communication links and hardware infrastructure must satisfy to ensure correct functionality of the network.

Starting from sensing requirements and an abstraction of the physical components a first version of the network topology is generated. Specifically, virtual components are replaced by an adequate number of physical components that ensure enough sensing, actuating, and control capabilities.

Starting from this network topology and the requirements on latency and bit error rates, routing and MAC algorithms are designed to optimize for power consumption. At this stage, the topology may be modified and extra physical components may be added to improve power performance and robustness of the communication infrastructure. This implies the creation of a library of routing and MAC algorithms whose performance can be modeled using protocol level tools (i.e. VisualSense or Omnet++).

When the final topology and protocol stack are determined, the functionality of the application can be mapped into the network architecture.

This paper is focused on the specification and translation into requirements for the network architecture. How the remaining steps can be approached is discussed in [11].

2. RIALTO MODEL

Following the approach of [1], in the proposed framework, designers describe the application in a Rialto Model in terms of Virtual Controllers, Virtual Sensors, and Virtual Actuators.

Tokens. Virtual components exchange data Tokens that are the abstraction of queries and commands. A query specifies the attribute to be sensed (i.e. vibration, temperature), the function to return (e.g., all the data values, average data value), the sampling rate, and the time scope. The time scope defines the interval of time for which the sensing should be performed. A command specifies the type of actuation (e.g., switch off the robot, increase temperature), the intensity of the actuation (i.e. temperature should be increased of 5 degrees Celsius), the need for an acknowledgment, and the time scope of the actuation. Both queries and commands also specify a tolerated latency and message error rate for the communication.

A token has nine fields, and its structure is:

$$Token = (q, c, n, a, v, Ti, Tf, L, Q),$$

where:

$q \in \{0, 1\}$ specifies if it is a query or a command, $c \in \{0, 1\}$ specifies if it is a request or a response, n is the function to return for a query or the need for an acknowledgment for a command, a is the attribute of the query or the type of actuation. v is the required sampling rate (for a query) or the intensity of the actuation (for a command), T_i and T_f are respectively the beginning and the end of the scope of the query (i.e. "Give me humidity data from time T_i to time T_f"), L [sec] is the latency requirement, Q is the quality of service requirement (bit error rate).

Virtual Controller. A *Virtual Controller (VC)* contains the description of the control algorithm of the application. If the application has more than one independent control algorithm, multiple Virtual Controllers have to be specified. The VC is only an abstraction of the control capabilities required by the application. This abstraction does not restrict our design space to a centralized control solution. In fact, in the physical implementation, the control algorithm described in a single VC could be implemented in a distributed fashion whenever it is convenient. Similarly, the functionalities of different Virtual Controllers could be implemented in the same physical component. Usually, designers already have

a good idea of where the physical controller, or controllers, can be placed. Consequently, they can embed this location information into the VC and limit the design space space exploration. The internal structure of a VC is a cyclic control routine. The number of queries and commands that can be generated during a control cycle must be limited. Consequently, no while loop with a query or command inside is allowed within a control cycle. Furthermore, the user can specify the time scope of the control cycle (how much time between two consecutive executions). If such parameter is not specified, we assume that the time scope is given by the the lowest T_i and the highest T_f of the generated queries or commands.

Virtual Sensor and Virtual Actuator. A *Virtual Sensor (VS)* represents a sensing area. This abstraction is useful because designers know which are the areas that need to be sensed, but they generally don't know how many sensors must be placed to cover that area and how they have to placed. Similarly to the VC, there is not necessarily a one-to-one relationship between virtual sensors and physical sensors. The number and the type of physical sensors that will be used to implement a virtual sensor is an implementation choice.

A *Virtual Actuator (VA)* represents an actuation capability. Similarly to the VS, the user describes the position of the VA, but the number and type of physical actuators that will be selected to implement its functionality is an implementation choice.

VA and VS are sequential threads of computation. They read the queries (commands) at their inputs, perform their sensing (actuating) task to satisfy those requests, and return data (if necessary) to the controller that sent the query (command). They are composed of two main functions: "Evaluate Inputs" and "Task". The "Evaluate Inputs" function specifies the read semantics of the actor, while the "Task" function specifies the method and the accuracy with which the actor fulfills the required sensing/actuation task.

Communication Media. Actors communicate through bidirectional, lossless, unbounded FIFO channels. Each channel is characterized by two separated queues, one for each direction. Connections are allowed only between a VC and a VA, and between a VC and a VS. Consequently, no connection is allowed between two Virtual Sensors, two Virtual Actuators, or a Virtual Sensor and a Virtual Actuator. This restriction makes sense because we are describing an application using logical components. Connections between two sensors (commonly refered to as multi-hopping) are an implementation option, and as such they don't belong to the application description level of abstraction. Similarly, a connection between two physical controllers is an implementation option, but at the application description level connections between two Virtual Controllers are not allowed. Hence, if a Virtual Controller needs a particular set of data, it has to send a query directly to a Virtual Sensor.

3. RIALTONET

After the application is described, the description is translated into an internal representation called RialtoNet.

The goal of RialtoNet is to generate a set of requirements to design a sensing and communication infrastructure that is able to satisfy every possible request of the controlling algorithms. Consequently, we need to evaluate all the various combinations of requests that Virtual Controllers could generate. Since the number of these combinations in a control routine is typically limited, this exhaustive state space exploration is often very manageable.

Generation of the RialtoNet. The generation of the RialtoNet goes through the following steps:

1. A set of actors called VCBranches is generated from each VC. We consider the conditional branches in a single control cycle. For each conditional branch that involves the possibility of sending a request, we consider both scenarios: the one in which the branch is taken, and the one in which the branch is not taken. This analysis generates all the possible combinations of queries and commands within a single cycle. Each of these combinations generates a VCB. A VCB is composed by a sequence of "SEND" instructions that represent a possible combination. Consequently, the VCB is an actor that is only able to send a predetermined sequence of tokens ("source" actor). Since in the control cycle of the original VC code there is only a limited amount of queries and commands, also the number of "SEND" instructions in a VCB is limited.

2. An actor called Virtual Sense Skeleton (VSS) is generated from each VS, and an actor called Virtual Actuator Skeleton (VAS) is generated from each VA. The VSS and VAS are sequential threads of computation. Similarly to the VCB, the VSS and VAS do not inherit the information regarding read and write semantics from their originating actor. They are composed by a "Task" function that is fired whenever their firing rules are satisfied. The "Task" code is inherited from their generating VS or VA. VSS and VAS have an internal variable called *Progression Tag*. As we will show in the next Section, this variable indicates the end of the time scope of the last query or command that has been served.

3. An extra actor, called Sink, is generated. The Sink has only input channels and it is used to store the results of a RialtoNet execution.

4. These actors are connected together to form a RialtoNet. Actors communicate through unbounded, unidirectional, lossless, FIFO channels. Each VCB inherits the connections of its generating VC in the Rialto Model. The direction of these connections is from the VCB to the VSS or VAS. Each VSS and each VAS has an output connection to the Sink.

Execution of the RialtoNet. Before describing the read and write semantics, we need to introduce the END Token, a particular token that is automatically produced by a VCB to all its output channels upon termination of its sequence of "SEND" instructions, or by a VSS or VAS to the Sink whenever its execution is terminated. Its structure can be interpreted as:

$$END = (q, 0, 0, 0, 0, null, \infty, null, null)$$

The execution of the RialtoNet is based on a model of computation that takes inspiration from Kahn Process Networks (KPN) [2, 3]. The VCB follows a non-blocking write semantics. Since it is a source actor, no reading semantics needs to be specified. The Sink has a non-blocking read

semantics. The VSS and VAS have blocking read and non-blocking write semantics. Since the blocking read rules for VSS and VAS are the same, we explain them only for the case of the VSS.

1. The VSS stalls its execution until all its input queues have at least one token. When all the input queues are non empty, the VSS evaluates the first token of each of the input queues.

2. If a VSS has END tokens in all its input queues, it sends an END token to the Sink and stops executing.

3. Otherwise, the VSS selects the token with lowest T_f. If more than one token happens to have the same T_f and it is the lowest, all of these tokens are selected. Consequently, an END token is never consumed because it has ∞ in its T_f field.

4. The VSS fires its sensing task and produces a set of requirements. These requirements are obtained by performing a logical AND of the "a" (attribute to sense) and "v" (sensing rate) fields of all the input tokens whose T_i field is less than or equal to the T_f field of the selected token. This information is embedded into a "Requirement Token" that is sent to the Sink.

5. The VSS updates its Progression Tag to the value of the T_f field of the selected token. The selected token is consumed, i.e. it is removed from its input queue and destroyed.

Requirement on Queries. Queries sent in the same VCB connection must have non overlapping time scopes. This is to avoid the situation in which, after advancing to serve a query, a VS would have to backtrack to serve another query with different requirements.

Queries emitted from the same VC branch must have non decreasing T_f field. This is to avoid the phenomenon of "sending a query to the past".

Termination and Requirements Generation. The execution terminates when each VSS and each VAS has sent an END token to the Sink. The Sink displays the received requirement tokens that specify the sensing and the communication requirements that the network architecture has to satisfy for each time-scope.

Properties. The Execution of the RialtoNet is based on a deterministic MoC. Because of the blocking read mechanism, VSS and VAS are continuous functions under the pointwise prefix order. Furthermore, VCBranches and Sink are also continuous functions. Consequently, the network is a composition of continuous functions and it has only one behavior.

Another important property of the RialtoNet execution is that it does not deadlock. Deadlock may happen only if a VSS or VAS waits in vain for a token that will never arrive. The introduction of the END token is tailored to avoid this problem. The idea of introducing the END query to resolve unwanted deadlocks can be seen as a particular case of the "null" message introduced by Misra in [13] when dealing with asynchronous parallel simulations.

The RialtoNet does not follow the read and write semantic specified in the original Rialto Model. It is only an internal representation that is generated to efficiently organize the analysis of the quantities that are of interest to set the requirements on the communication links and sensing capability of the physical network. Consequently, the RialtoNet is not used to check the functionality of the application specified in the Rialto Model. However, the choice leaves complete freedom to the application designer to speficy the control algorithm with the semantic and yet be able to derive information to build an appropriate network architecture.

4. CONCLUSIONS

We presented Rialto, a framework for capturing control-based WSN applications and generating a set of constraints on latency and sensing capabilities that the communication and sensor infrastructure must satisfy.

Rialto is based on two steps. First the application is described in terms of virtual components, queries and commands in a Rialto Model. Then this representation is translated into an internal format called RialtoNet the allows for a complete exploration of all the possible combination of requests that the WSN should serve. The RialtoNet is executed and a set of requirements for the design of the communication protocol and sensing capabilities is generated.

Rialto allows the user to express the application with a high degree of freedom and to march towards implementation with a correct-by-construction methodology.

5. REFERENCES

[1] M. Sgroi et al., "A Service-Based Universal Application Interface for Ad-hoc Wireless Sensor Networks", whitepaper, U.C.Berkeley 2004.

[2] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," Proc. of the IFIP Congress 74, North-Holland Pub.

[3] G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," Information Processing 77, B. Gilchrist, editor, North-Holland Publishing Co., 1977.

[4] http://www.motion.aptd.nist.gov/

[5] J. Hill, D. Culler, "Mica: A Wireless Platform for Deeply Embedded Networks" IEEE Micro., vol22 (6), Nov/Dec 2002, pp.12-24.

[6] D. Gay et al., "The nesC Language: A Holistic Approach to Networked Embedded Systems", Proceedings of Programming Language Design and Implementation (PLDI) 2003, June 2003.

[7] P. Levis et al., "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Application",SENSYS 03.

[8] A. Varga, "The OMNeT++ Discrete Event Simulation System", in European Simulation Multiconference June 2001.

[9] P. Baldwin et al., "Visualsense: Visual Modeling for Wireless and Sensor Network Systems", UCB ERL Memorandum UCB/ERL M04/8 April 23, 2004.

[10] http://ptolemy.eecs.berkeley.edu

[11] R.C. Shah et al., "Joint Optimization of a Protocol Stack for Sensor Networks", MILCOM 2004.

[12] E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation", IEEE Transactions on CAD, 17(12), December, 1998.

[13] J. Misra, "Distributed Discrete-Event Simulation",ACM Computing Surveys, Vol. 18, No. 1, 1986, pp. 39-65.

[14] Y. Yu et al., "Communication Models for Algorithm Design in Wireless Sensor Networks", IPDPS '05.

[15] A. Bakshi, V.K. Prasanna, "Algorithm Design and Synthesis for Wireless Sensor Networks", ICPP '04.