UNIVERSITÀ DEGLI STUDI DI PALERMO

Dottorato in Ingegneria dell'Innovazione Tecnologica
Dipartimento dell'Innovazione Industriale e Digitale (DIID)
ING-INF/05

SYMBOLIC PROGRAMMING OF DISTRIBUTED CYBER-PHYSICAL SYSTEMS

IL DOTTORE
GLORIA MARTORELLA

IL COORDINATORE
CH.MO PROF. SALVATORE GAGLIO

IL TUTOR
CH.MO PROF. SALVATORE GAGLIO

CO-TUTOR
PROF. DANIELE PERI

CICLO XXIX
ANNO CONSEGUIMENTO TITOLO 2017

To my family

Abstract

Cyber-Physical Systems (CPSs) tightly integrate physical world phenomena and cyber aspects of computational units. The composition of physical, computational and communication systems demands different levels and types of abstraction as well as novel programming methodologies allowing for homogeneous programming, knowledge representation and exchange on heterogeneous devices.

Current modeling approaches, frameworks and architectures result fairly inadequate to the task, especially when resource-constrained devices are involved.

This work proposes symbolic computation as an effective solution to program resource constrained CPS devices with code maintaining strict ties to high-level specifications expressed in natural language while supporting interoperability among heterogeneous devices. Design, architectural, programming, and deployment aspects of CPSs are addressed through a single formalism unifying the specification of both cyber and physical parts of CPSs. In particular, programming patterns are modeled as sequences of words adhering to natural language syntax and semantics. Given a software under test (SUT), i.e. an input program expressed as a natural language sentence, formal specifications are used to generate oracles for sentence verification and to generate input test cases. The choice of natural language inspired programming supplies a mechanism for the development of the same software on different hardware platforms, ensuring interoperability among heterogeneous devices. Formal specifications also permit to generate stress tests in order to verify that program components behave as expected in repeated execution.

In order to make high-level symbolic programs run on real hardware devices with no loss of expressivity during the translation of high-level specifications into an executable implementation, this work proposes a novel software architecture, Dis-

tributed Computing for Constrained Devices (DC4CD), as a supporting platform. The proposed architecture enables symbolic processing and distributed computing on devices with very limited energy, communication and processing capabilities that can be integrated into CPSs.

In particular, DC4CD has been extensively used to test the symbolic distributed programming methodology on Wireless Sensor Networks (WSNs) that include nodes with actuation abilities. The platform offers networking abstractions for the exchange of symbolic code among peer devices and allows designers to change at runtime, even wirelessly on deployed nodes, not only the application code but also system code.

Acknowledgments

First and foremost I would like to express my sincere gratitude to Prof. Salvatore Gaglio for his constant guidance and precious advice throughout the Ph.D course. It has been a wonderful opportunity and an honor for me to complete my Ph.D thesis under his supervision.

I am deeply grateful to Dr. Daniele Peri for all the ideas, scientific advice, and patience in guiding me through this research project. Without his constant trust and gentle encouragement, this thesis would not have been possible. His insightful suggestions, inspiration and guidance have been invaluable to make my Ph.D experience productive and stimulating.

I would also like to thank Prof. Giuseppe Lo Re. His technical discussions and precious suggestions have been very important for my personal and professional development.

I am indebted to all my colleagues in Networking and Distributed Systems Research Group. A special mention of thanks goes to Pierluca Ferraro, Vincenzo Agate, Davide Vassallo, and Antonino Piazza for creating and maintaining an exciting working atmosphere in which mutual support, discussions and honest friendship have been the key ingredients.

Finally, I will forever be grateful to my family and my husband for their continous encouragement, endless love and blessings.

Contents

List of Figures

List of Tables

Glossary

ASVM	Application Specific Virtual Machine
CoAP	Constrained Application Protocol
REST	REpresentational State Transfer
IPv6	Internet Protocol version 6
PStg	Symbolic Permanent Storage
TStg	Symbolic Temporary Storage
PS	Parameter Stack
RS	Return Stack
DTC	Direct Threaded Code
ITC	Indirect Threaded Code
MCU	MicroController Unit
RISC	Reduced Instruction Set Computer
I/O	Input/Output
SPI	Serial Peripheral Interface
I2C	Inter-Integrated Circuit
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
PWM	Pulse Width Modulation
RF4CE	Radio Frequency for Consumer Electronics
6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
ISM	Industrial, Scientific and Medical (*Applications)*
GPIO	General-Purpose Input/Output
ADC	Analog-to-Digital Converter
DAC	Digital-to-Analog Converter

OS	Operating System
VM	Virtual Machine
MAC	Media Access Control
CRC	Cyclic Redundancy Check
HAL	Hardware Abstraction Layer
CTP	Collection Tree Protocol
TOS	Top Of Stack
IPS	Instruction Per Second

Chapter 1

Introduction

The need of integrating heterogeneous aspects coherently, the lack of high-level abstractions, and resource constraints of hardware platforms, are some of the key issues that make the development of applications for Cyber-Physical Systems (CPS) challenging (Simko et al., 2013).

Due to these stringent requirements, this work proposes a programming methodology based on a symbolic computational paradigm as a mean to make target code mantaining a close resemblance to the high-level specification of the system, while fostering interoperability among heterogeneous devices.

From a high-level perspective, CPSs formal specification and verification is challenging (Simko et al., 2014). Much of the existing literature especially focuses on testing automation, in order to speed the verification process by addressing the so called *oracle problem*. A test oracle is a mechanism allowing to determine the correct behaviour for all of the given inputs provided to a system (Barr et al., 2015). However, there has been much interest and recent advances concerning the automatic generation of test inputs, while the problem of automating the oracle generation remains an under explored aspect which is the focus of current research investigations (Harman et al., 2013).

In this direction, this work proposes a rule-based system that compenetrates physical quantities, computational, functional and domain aspects to support the development of CPS applications using high-level programming patterns inspired by the syntactic and semantic structure of natural language sentences. The sys-

tem supports runtime verification of software components, i.e. sentences, through automated oracle and test case generation. It also provides supports for automatic generation of stress tests on the target hardware platform.

Moreover, even if a high-level description of the functional behavior is provided that seems clear in natural language, it is not usually so in the programming language of choice for the mainstream development platforms (Lai and Jirachiefpattana, 2013). A set of guidelines are thus presented to attain symbolic code that is easily understandable and resembling as much as possible a natural language description.

However, due to resource constraints, which distinguish the vast majority of CPS devices, e.g. Wireless Sensor Network (WSN) nodes, providing abstractions that can be used to incorporate high-level processing of more structured and symbolic data than those resulting from mere logging of a few key physical quantities may prove impractical with the standard programming methodologies (Patel and Cassou, 2015).

To mitigate this issue, a novel software platform was developed in the course of this study that permits to execute symbolic programs as sequences of high-level natural language words, to define procedures and services according to the application target, and simultaneously test them. Nodes can be reprogrammed when needed, even after network deployment. Even though the software platform currently targets the IRIS mote hardware, due to the high-level programming approach, porting it to other hardware is straightforward. A simple but powerful abstraction that the platform supports is the executable symbolic code exchange among nodes. This mechanism, while abstracted, is implemented at a very low level avoiding the burden of a complex and thick software layer between the hardware and the application code.

1.1 Motivations and Goals

CPSs are heterogeneous systems resulting from a tight integration between the physical world phenomena and the cyber aspects of computational units. Developing CPS applications compels the programmer to embrace networking and computational, functional and domain aspects into a coherent whole (Khaitan and

McCalley, 2015). Such an integration is especially difficult when dealing with Ambient Intelligence (AmI) and Internet of Things (IoT) applications, which strongly demand novel programming paradigms and high-level abstractions to program heterogeneous devices in a homogeneous way in order to foster interoperability (Patel and Cassou, 2015).

These requirements collide with the scarcity of available resources on embedded and CPS devices. For instance, many approaches found in literature adopt the Java Virtual Machine (JVM) to bring high-level symbolic programming to this class of computational units. However, this choice often forces to sacrifice features for acceptable resource consumption, leading to partial implementations (Brouwers et al., 2009a). More in general, high-level languages are not easily applicable to CPS programming and resource constraints are the major issue in developing and debugging CPSs (Zheng and Julien, 2015).

This work proposes a symbolic computational paradigm to program heterogeneous CPS devices through sequences of high-level words whose syntax and semantics can be made as similar as possible to natural language sentences, thus allowing to develop the same program on different hardware platforms. In particular this dissertation especially focuses on resource-constrained CPS devices, as they currently represent the major hinder to device integration and interoperability.

The application scenario this work refers to, is composed of several objects, e.g. sensors and actuators, pervasively deployed in the environment, which are provided with computational abilities and interact with the physical world.

To model these kind of high-level application domains, in this dissertation, a rule-based system is proposed which integrates knowledge about Cyber-Physical concepts and properties, hardware components and programming patterns. In particular, programming patterns are inspired by natural language. The system also provides an automated oracle for a given software under test (SUT). The SUT and the respective oracle are then executed on the target platform using the proposed symbolic computational paradigm. The system is also able to generate test cases and stress test code.

The rationale behind the choice to focus on natural language patterns lays not only in the reduction of intermediate abstraction layers to map CPS concepts into executable code but also in programming practice.

This represents an alternative to mainstream practices in which the abstract model passes through a series of intermediate representations that must then be translated into the target programming language (Martins and de Almeida Falbo, 2008; Wada et al., 2007; Shahbaz et al., 2011). During this translation process, either automatic or not, the semantic clarity and the expressiveness of the high level specification is progressively lost.

In order to reduce the decoupling between specification and execution, and the loss of expressivity during the translation process, this work addresses the implementation of executable specifications that are semantically clear and easily verifiable, and provides guidelines and key steps to incorporate semantics into the practical development, given an initial specification. Data and code are thus represented through high-level symbols associated with an obvious meaning in natural language and in the task knowledge domain. In fact, the adopted symbolic computational paradigm encourages the programmer to build programs as sequences of natural language words. Indeed, code retains a close resemblance to the functional description of the system. This makes it simple translating the specifications into executable code and the code back again into specifications.

Target environments should also be lightweight enough to be run on resource-constrained sensor devices composing the CPS ecosystem. Wireless Sensor Networks (WSNs) are examples of the latter, being composed of tiny wirelessly interconnected sensor nodes that are equipped with a microcontroller, a radio interface subsystem, some sensor devices and an autonomous power supply, usually consisting in batteries. Generally, such devices are characterized by quite constrained resources in terms of energy, communication and processing capabilities. WSNs represent a very active research area as several applications have been proposed in literature in several contexts such as biomedical, healthcare, military, industrial and environmental fields (Akyildiz et al., 2002).

However, to cope with the inner limitations of nodes, most WSN designs use nodes as mere tools to sense physical quantities (Rawat et al., 2014), while the application logic is implemented in a centralized fashion or in the Cloud (Kovatsch et al., 2012). Architectural efforts are thus required to advance the ordinary use of WSN nodes, as well as similar resource constrained devices, to support symbolic distributed computing, enabling advanced applications that are able to reason lo-

cally to avoid the latency and network overhead issues of centralized designs (Antola et al., 2014) and overcoming limitations of nodes (Xu et al., 2014).

Current approaches mainly focus on node reprogramming through either remote update of binary images or some high-level language interpretation. In the first case, remote updates of code require recompilation and subsequent distribution of the entire binary image. This involves large data transfers that are only marginally mitigated by binary incremental update approaches (Miller and Poellabauer, 2010; Munawar et al., 2010). Encoding and compression of binary data (Deng and Yang, 2012) as well as modular platforms supporting incremental updates (Dong et al., 2011, 2013) have been proposed to reduce the size of the binary code in order to facilitate its distribution (Leligou et al., 2011; Chu et al., 2013; Dong et al., 2014).

The complementary approaches based on high-level language interpretation mostly propose using bytecode-based virtual machines implemented on top of some general-purpose WSN operating system (Oliver et al., 2014). Virtual machines simplify the deployment of the application software updates due to the portability of the bytecode with respect to binary images (Levis and Culler, 2002). However, code replacement cannot affect the operating system or the virtual machine itself. Moreover, although the use of bytecode may result in smaller application code, the expressiveness of the code itself is actually reduced (Alippi et al., 2011).

More importantly, interpreters for widespread high-level languages, such as Java and Python, require considerable amount of resources. As a prominent example, memory, which amounts to a few hundreds kilobytes in the class of resource constrained devices, is almost saturated by the rather high memory footprint of these interpreters. The development of sophisticated applications becomes thus particularly difficult or unfeasible (Alessandrelli et al., 2013).

Finally, existing architectures that are based on interpretation do not include direct support for distributed programming. From an architectural point of view, this would require further design efforts to incorporate joint mechanisms for homogeneous information representation on heterogeneous platforms as well as for the exchange of such information among different devices (Cecilio and Furtado, 2014). Furthermore, current implementations support cooperation at the application level by statically setting in advance all the possible message formats.

Virtual machines that directly execute high level symbolic code preserve expressiveness without compromising compactness, since many operations can be expressed using few robust constructs (Evers et al., 2007a). Also in this case, virtual machines are built above a general-purpose operating system (Evers et al., 2007b).

As a first step toward interoperable and symbolically programmable CPSs this dissertation introduces a novel software architecture, Distributed Computing for Constrained Devices (DC4CD), primarily designed to enable distributed computing and symbolic processing on small-scale interconnected devices with limited resources. DC4CD exploits the interpretation and distribution of code as plain text strings, rather than as bytecode. This choice, while not sacrificing code compactness, provides a straightforward way to distributed symbolic processing. Moreover, it abstracts the characteristics of the target hardware enhancing the interoperability between heterogeneous devices.

As in the case of Active Networks (Stehr and Talcott, 2004; Levis et al., 2005a), DC4CD supports code injection to remote devices, but the interpreter does not run above a thick layered architecture, so that the detachment of the application from the hardware is avoided. The injected code can trigger remote execution of local or distributed behaviors as well as modifications of the remote program memory. Finally, assembly code can also be distributed, again in the form of text strings, to avoid the interpretation overhead in time-critical code sections.

DC4CD includes high level symbols for the injection of symbolic code among entities to support the development of collaborative in-network processing applications. Indeed, a distributed application can be thought in terms of exchanges of symbolic code rather than of predefined format messages that the nodes must be programmed to handle in advance. The exchange of symbolic code among nodes is suitable for the implementation of in-network distributed processing as well as for the inclusion of context-aware knowledge while overcoming interoperability issues.

1.2 Contributions

The main contributions of this dissertation are:

- A methodology to develop meaningful implementations, e.g. abstract models, that are strictly coupled to the specifications and are executable on resource constrained target devices; the main goal is reducing the abstraction layers that are required to map CPS concepts into executable code. Symbolic computation is proposed as an effective way (i) to program heterogeneous devices in an homogeneous way, (ii) to develop programs whose syntax and semantics are similar as possible to natural language and (iii) to incorporate properties of the physical world in the runtime verification process on target devices.

- A rule-based system that incorporates computational, functional and domain aspects to support the development of CPS using high-level programming patterns that preserve the syntactic and semantic properties of natural language sentences. The system supports runtime verification of software component through automated oracle and test case generation. It is also able to build stress tests to highlight software issues arising during repeated execution of software components on the target hardware.

- The design and development of a novel software architecture, Distributed Computing for Constrained Devices (DC4CD), able to process symbolic programs as sequences of high-level words and to enable distributed computing even on resource-constrained devices. The platform integrates the functionalities of a high-level symbolic interpreter, a compiler, and an operating system, and includes networking abstractions to exchange high-level symbolic code among peer devices.

1.3 Dissertation Outline

The remainder of the dissertation is organized as follows.

Chapter 2 describes the knowledge-based system for the development of CPS applications as sequences of words matching natural language patterns. In particular, the oracle and test generation for runtime verification of CPS applications is presented, as well as the generation of stress test code to verify that program

components behave as expected in repeated execution. As the gap between specifications and the resulting implementation in the chosen programming language is notably a source of errors in CPS design, Chapter 3 discusses how to cope with these issues by defining guidelines to designers in order to develop meaningful and executable specifications, running effectively on the target hardware, through a symbolic computational paradigm. Concretely, the methodology has been applied to specify the functional behavior of a radio transceiver chip. Chapter 4 presents Distributed Computing for Constrained Devices (DC4CD), a novel software architecture that supports symbolic distributed computing on resource-constrained devices, such as Wireless Sensor Network nodes. The Chapter details the computational paradigm of the proposed architecture, which is able to process sequences of high-level words, and describes the platform design. Moreover, a simplified case study and an experimental evaluation of the proposed architecture are presented. Chapter 5 applies the proposed metodology and platform to develop complex applications in different real scenarios. Finally, Chapter 6 states some conclusion about this work.

1.4 Publications

Parts of the work in this thesis have been published in several referred conference proceedings and book chapters:

- A Symbolic Distributed Event Detection Scheme for Wireless Sensor Networks. S. Gaglio, G. Lo Re, G. Martorella, D. Peri. *In Proceedings of the 21th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 16).*
- Use of Forth to Enable Distributed Processing on Wireless Sensor Networks. S. Gaglio, G. Lo Re, G. Martorella, D. Peri. *In Proceedings of the 31th EuroForth Conference (EuroForth2015).*
- High-level Programming and Symbolic Reasoning on IoT Resource Constrained Devices. S. Gaglio, G. Lo Re, G. Martorella, D. Peri. *In EAI Endorsed Trans. Cognitive Communication 1(2): e6 (2015).*

- Closing the Sensing-Reasoning-Actuating Loop in Resource-constrained WSANs through Distributed Symbolic Processing. S. Gaglio, G. Lo Re, G. Martorella, D. Peri, S.D. Vassallo. *In Proceedings of the 20th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 15).*

- Programming distributed applications with symbolic reasoning on WSNs. S. Gaglio, G. Lo Re, G. Martorella, D. Peri. *In Proceedings of the 2015 International Conference on Computing, Networking and Communications (ICNC).*

- Development of an IoT Environmental Monitoring Application with a Novel Middleware for Resource Constrained Devices. S. Gaglio, G. Lo Re, G. Martorella, D. Peri, S.D. Vassallo. *In Proceedings of the 2nd Conference on Mobile and Information Technologies in Medicine (MobileMed 2014).*

- High-level Programming and Symbolic Reasoning on IoT Resource Constrained Devices. S. Gaglio, G. Lo Re, G. Martorella, D. Peri. *In Proceedings of the First International Conference on Cognitive Internet of Things Technologies (COIOTE 2014).*

- A Fast and Interactive Approach to Application Development on Wireless Sensor and Actuator Networks. S. Gaglio, G. Lo Re, G. Martorella, D. Peri. *In Proceedings of the 19th International Conference on Emerging Technologies and Factory Automation (ETFA2014).*

- A Lightweight Middleware Platform for Distributed Computing on Wireless Sensor Networks. S. Gaglio, G. Lo Re, G. Martorella, D. Peri. *In Proceedings of the 2nd International Workshop on Body Area Sensor Networks (BASNet-2014).*

- An AMI System for User Daily Routine Recognition and Prediction. S. Gaglio, G. Martorella. *Advances onto the Internet of Things, in Advances in Intelligent Systems and Computing series, Springer, pages 1-13, 2014.*

- Hardware and Software Platforms for Distributed Computing on Resources Constrained Devices. G. Martorella, D. Peri, E. Toscano. *Advances onto*

the Internet of Things, in Advances in Intelligent Systems and Computing series, Springer, pages 1-13, 2014.

Chapter 2

Natural Language Programming Patterns for Cyber-Physical Systems - Design and Verification

CPSs are the integration of computational and physical systems. In fact, physical world events and its dynamics are perceived by sensors and internally converted into some hardware representation. Complementary, processing tasks undertaken by computational devices affect the physical environment through actions perfomed by actuators (Khaitan and McCalley, 2015).

In this context, software testing including runtime verification mechanisms, such as those based on test oracles, on-board resource-constrained CPS devices demands specific efforts (Iyenghar et al., 2013; Barr et al., 2015).

A test oracle is a method to obtain expected results about the behavior of a system for a given input provided to the system to be tested. In the absence of automated oracle generation, determining whether observed behavior is correct remains a human prerogative (Harman et al., 2013).

Assertion-based (Araujo et al., 2011), model-based (Adalid et al., 2014; Li and Offutt, 2014) and fault-injection (Ghosh and Kelly, 2008) approaches have been proposed to address the problem of test oracle generation for the Java Virtual Machine (JVM). Other works focus on test case generation techniques specifically intended for embedded systems (Hasanain et al., 2015; Iqbal et al., 2015; Yu et al.,

2014; Iyenghar et al., 2010). The problem of verifying the correctness of high-level symbolic expressions is addressed, even if not specifically for either embedded or runtime applications, with grammar-based methodologies enriched by denotational semantics (Guo, 2016).

In this Chapter a rule-based system integrates knowledge about Cyber-Physical concepts and properties, hardware components and programming patterns, and provides an automated oracle for a given software under test (SUT). In the proposed approach, a SUT consists of a software component, i.e. a natural language sentence, while a test oracle is a program, expressed as a sequence of low-level symbols, allowing to inspect the hardware state and to verify whether the system behaves correctly. The SUT and the respective oracle are then executed on the target platform using a symbolic computational paradigm. The system is also able to generate test cases and stress test code.

The chapter is organized as follows. Section 2.1 describes the key features of the symbolic computational paradigm, while the description of the proposed rule-based system is given in Section 2.2. In Section 3.6 it is shown how the system can be used to generate stress tests for the target hardware also providing experimental results, while Section 2.4 concludes the chapter.

2.1 The Semantic Model

High-level applications for CPS typically refer to the scenario shown in Figure 2.1. Several objects, either sensors and actuators, related to domain of interest are provided with computational abilities and interact with the environment.

In this context, a symbolic programming paradigm gives the designer the possibility to develop code that is aligned to natural language syntax and semantics.

The proposed approach exploits a simple stack-based symbolic computational model (see Appendix) that can be effectively implemented even on resource-constrained devices (Rather et al., 1993) as detailed in Chapter 4. The rationale behind this choice is provided in Chapter 3.

Symbols are meaningful enough to define a Domain-Specific Language (DSL) while their execution directly affects the bare hardware.

Henceforth the terms symbol and word will be used as interchangeably.

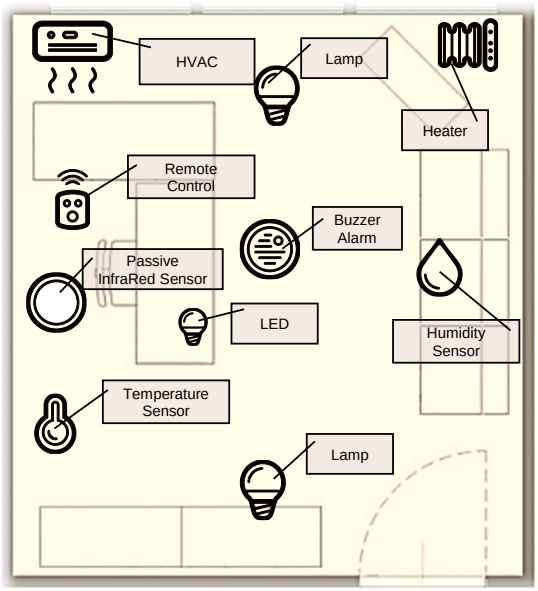


Figure 2.1: A possible CPS reference application scenario.

As an example, a CPS is considered that includes an actuation device, such as a lamp, connected to a relay on-board the target hardware platform. A simple program to switch the lamp on could be:

```
lamp on
```

The goal expressed through this particularly clear semantics is reached, according to the computational paradigm, by executing the two syntactically independent words `lamp` and `on`, one after another, on the target machine.

Due to the close bond between the cyber and physical world, physical world actions reflects on the cyber world through hardware state changes. From the hardware perspective, switching the lamp on implies driving the relay to close the circuit. Relays can be either normally-open or normally-closed and in some I/O expansion boards both kinds are provided. This two types are usually indicated in technical documentation with the abbreviations *no* and *nc*.

However, the syntax, semantics and implementation of the high-level program, e.g. the computation associated to each word, is left to the programmer. Hence, in such a case, the execution of the word `lamp` could be expressed in terms of other words describing lower level operations as in the following word sequence:

```
no relay
```

The programmer may design these words using a top-down approach, so that they incorporate the low-level details, including the communication protocol over the specific hardware interface (GPIO, SPI, I2C and so on). Finally, the word `on` finally switches the lamp on and drives the proper relay. The same syntactic pattern could be used to switch other devices on or off, for instance an LED:

```
led on
```

This would only require defining the appropriate hardware-related words for the LEDs in question.

Due to the inherent complexity arising from not relying on a predefined syntax, the oracle problem for the symbolic paradigm provides a wide range of possible verification activities that go beyond mere input generation for unit testing and syntactic verification. An oracle, in this paradigm, is thus a sequence of words that: (i) is syntactically and semantically valid, and (ii) verifies that the state of the target hardware and the physical environment are the expected ones after the execution of the SUT.

Generation of oracles and test cases is detailed in the next section.

2.2 System Overview

The proposed system integrates knowledge about the Cyber-Physical domain, such as physical quantities and cause-effect relations, hardware specifications as well as syntax and semantics modeling valid word sequences adhering to programming patterns inspired by natural language (Figure 2.2). Rules defining Cyber-Physical concepts and natural language patterns are not tied to a specific application. Application code is instead strictly dependent on hardware specifications. However, the latter can be used to test all the SUTs running on the same hardware.

The Physical World domain is bidirectionally mapped to the internal hardware representation through sensing and actuation devices, as shown in Figure 2.3.

Given the three distinct, but closely tied components of the knowledge base the system gets the SUT as input and generates the oracle, if it exists, to be

incorporated into the SUT for its verification (see Figure 2.4). The input program is written by the programmer as a sequence of high-level words referring to the application domain. Similarly, the oracle is defined as a sequence of words that verifies the hardware effects during the execution of the test.

Due to the nature of CPS applications, it is possible to test each actuating and sensing sentence as independently verifiable components. The evaluation mechanism is that of a stack machine following the Reverse Polish Notation (RPN). For instance, in the following code:

```
temperature
50 < if
      green led on
   else
      green led off
then
```

the sensing sentence `temperature` writes the value of the physical quantity to the top of the stack (TOS), then a comparison with a constant threshold value determines which of the actuation sentences is to be executed. The three sentences can be verified alone, and the correctness of the three components can also be assessed through stress tests over a number of repetitions (see Section 3.6).

Finally, given a programmer-defined word set, the system is able to generate test cases by defining word sequences that adhere to natural language programming patterns specified in the knowledge base. Test cases and the respective oracles are then executed on the target platform implementing the symbolic computational paradigm.

The current implementation of the system is written in Prolog and targets a standard Forth environment (see Appendix). In the following, the three components of the knowledge base as well as the oracle and test case generation are detailed. Logic rules are presented in their Prolog embodiment.

2.2.1 Cyber-Physical Rules

The system includes the explicit specification of domain concepts and their relations. However, physical world features are confined to those provided by sensing

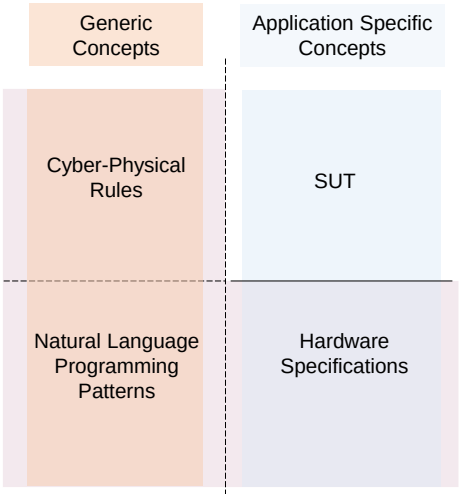


Figure 2.2: The three components of the knowledge base and the conceptual dependencies among them and the SUT.

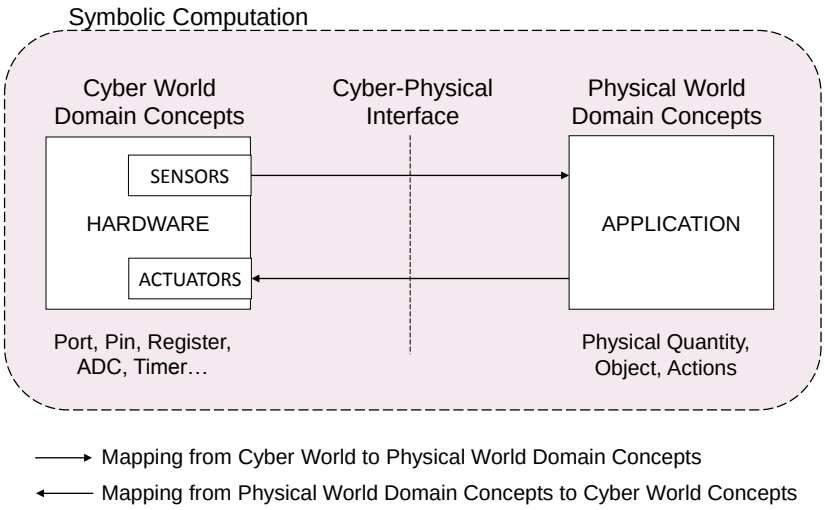


Figure 2.3: The Physical World domain is bidirectionally mapped to an internal hardware representation through sensing and actuation devices. Executable words of an application are defined in terms of lower level words concerning ports, registers, timers and other hardware components.

and actuation abilities of the hardware. In a broad sense, a CPS is specified in terms of its components, object classes and the interactions with the physical world through sensing and actuation devices, as shown in Figure 2.3.

The same symbols used by rules to specify CPSs can also be defined as ex-

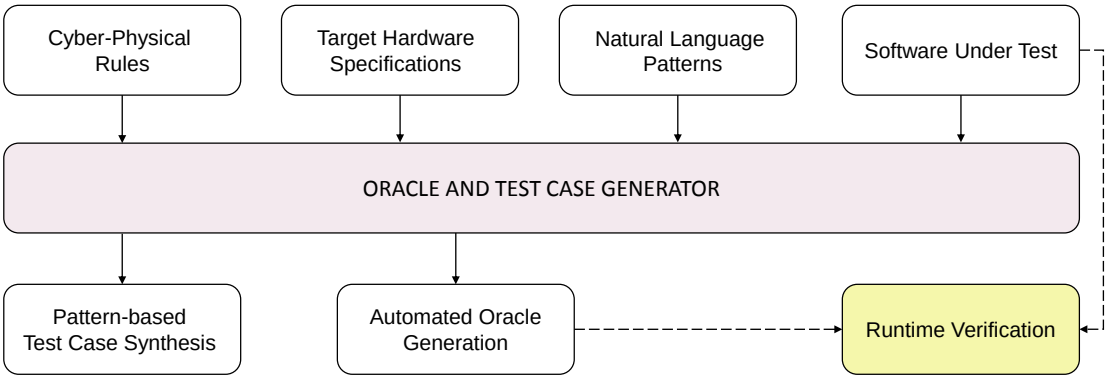


Figure 2.4: The system architecture of the proposed automated test case and oracle generation system for symbolic programming of Cyber-Physical Systems. The generated oracle is incorporated into the SUT for runtime verification on the target platform.

ecutable words. For instance, the word `lamp` not only defines a specific domain object, but is also a symbol that is used in the high-level code running on the target hardware.

The model including rules for the CPS scenario consists of specifications about:

- **Objects and their features**. Classes of objects composing the system e.g. lamps, LEDs, Heating Ventilation and Air Conditioning systems (HVAC), and so on, are defined as facts in the knowledge base. Object specifications also include class properties, possible states and relationships with physical world quantities. For instance, a restricted set of Cyber-Physical concepts includes the definition of domain objects and their possible states:

```
object(led).
object(sensor).
object(lamp).
state(led, on).
state(led, off).
inverted_state(off, on).
inverted_state(on, off).
emits(led,light).
```

```
emits(lamp,light).
emits(buzzer,sound).
emits(human,infrared).
```

LEDs having two states, on and off, are structural components of the system. Such a specification implicitly defines computational items composing the domain-specific language (DSL) used to write natural language-like code.

- **Properties of the physical environment**. Unlike the approaches found in literature (Derler et al., 2012), environmental features, i.e. physical quantities, are not described in terms of differential equations or continuous functions. Rather, ambient properties, such as physical quantities, are confined to those measurable by the defined sensor devices. This is explicitly formalized as follows:

```
physical_quantity(X):-
  object(S), instance(Z,S), sense(Z,X).
```

Knowledge about physical quantity properties can be also included as follows:

```
property(color,light).
property(frequency,sound).
```

- **Actuation**. High-level actions are described by means of their effects on the state of objects. As an example, a simple set of actuation rules is specified through the predicate `action(?Actuation_Sequence,?ObjectState)` as follows:

```
action([FinalState], [Object, _, FinalState]):-
                      state(Object, FinalState).

action([turn, FinalState], [Object, _, FinalState]):-
                      state(Object, FinalState).
```

```
action([change, state, to, FinalState],
[Object, _, FinalState]):-
                      state(Object, FinalState).

action([invert, state],
[Object, InitialState, FinalState]):-
                      state(Object, FinalState),
        inverted_state(FinalState, InitialState).

action([toggle, state],
[Object, InitialState, FinalState]):-
                    state(Object, FinalState),
     inverted_state(FinalState, InitialState).
```

Each action is declared by binding a list of words describing it in natural language to a list containing an object name along with the initial and final states of the action. Considering the above declarations, the simplest way to change the object state is to indicate the final state to be reached, as with the command `on`. This action is independent of the initial state, thus the universal quantifier *underscore* (`_`) is used in place of a variable. The same applies to the definition including `turn`. The other definitions depend instead on both states so they are explicitly evaluated through named variables. Alternatively, the rules include more complex natural-language sentences expressing sentences for possible actions having the same effects on the object state. Sentences are described that change (`turn on`, `change state to on`), or invert (`invert state`, `toggle state`) the state of an object. The definitions can be queried with either one or two instantiated arguments, to obtain the values satisfying the clause if existing. For instance, the clause:

```
action([turn, on], [led, _, on])
```

is true, as switching an LED object to the `on` state is a possible actuation action. Words composing sentences can be the same words used to build high-level sentences in the target programming language. Correct sentences

are those satisfying actuation programming patterns. Specification of programming patterns is discussed in Section 2.2.3.

- **Sensing**. Sensing allows to perceive the environment in term of its physical quantities by reading the appropriate sensor devices. Natural language commands for sensing are specified using the predicate `perception/2`, with the following template `perception(?Sensing_Sequence,?ObjectState)`, as such:

```
perception([Physical],[Object, _]):-
  sense(Object,Physical).
perception([read,value], [Object, _]):-
  sense(Object,_).
perception([query],[Object,_]):-
  sense(Object,_).
perception([get,value],[Object,_]):-
  sense(Object,_).
perception([get,sample],[Object,_];-
  sense(Object,_).
perception([sample], [Object, _]):-
  sense(Object,_).
perception([reading], [Object, _]):-
  sense(Object,_).
perception([value], [Object, _]):-
  sense(Object,_).
```

The simplest command to sense the environment consists in the name of the physical quantity to be sensed. More structured sensing operations include commands to read a value or query a sensor expressed as lists of words used in natural language sentences. Similarly, it is possible to specify commands for sensing by indicating perception properties like value type or range. For sake of simplicity, the definition above do not make use of this possibility that could be exploited, for instance, to test that `0` is a possible outcome value of the sensing operation `read value` performed on `sensor1`:

```
perception([read, value], [sensor1,0])
```

Similarly to actuation, syntactically valid sentences, which include the same words used to specify sensing commands, are those satisfying programming patterns for sensing, as discussed in Section 2.2.3.

- **Qualitative dynamics of the system**. Actuation actions affect the physical world state. These cause-effect relations are specified by rules that bind physical quantities to actuation abilities. This is specified through the predicate `effect(?Effect, ?ObjectState, ?Actuation_Sequence)`. For instance, turning a lamp on has the effect of increasing the ambient light:

```
effect([increased, light],
[Object, InitialState, FinalState], Action):-
FinalState=on, emits(Object, light),
state(Object, InitialState),
action(Action, [Object, InitialState, FinalState]).
```

on the contrary, turning a lamp off decreases the physical quantity:

```
effect([decreased, light],
[Object, InitialState, FinalState], Action):-
FinalState=off, emits(Object, light),
state(Object, InitialState),
action(Action, [Object, InitialState, FinalState]).
```

The knowledge base can be extended to include further objects, actions, perceptions, and events of the target application domain.

2.2.2 Hardware Specifications

Beside Cyber-Physical rules, the oracle generation engine requires the hardware configuration as input. From a structural point of view, the CPS is made of

hardware components that are instances of classes of objects. For a CPS provided with two LEDs, one lamp, two buzzers and three sensors, the device instances would be described as such:

```
instance(led0,led).
instance(led1,led).
instance(lamp1,lamp).
instance(buzzer1,buzzer).
instance(buzzer2,buzzer).
instance(sensor1,sensor).
instance(sensor2,sensor).
instance(sensor3,sensor).
```

A hardware device is uniquely identified by a label, typically the name it is given in the specifications or the schematics of the hardware. For instance, the hardware device named `led0` is clearly an LED. For sensor devices, the hardware specification also includes the physical quantity they measure:

```
sense(sensor1,light).
sense(sensor2,humidity).
sense(sensor3,infrared).
```

In this case, the CPS includes a light sensor, a humidity sensors and an infrared sensor. This description implicitly defines the physical quantities, which are considered as environmental properties of the CPS, as a whole. Furthermore, it is possible to qualitatively specify hardware device properties as attributes:

```
attribute(led0,green).
attribute(led1,red).
attribute(buzzer1,'4KHz').
attribute(buzzer2,'5KHz').
```

In this case, `led0` is a green LED, while `buzzer1` is a 4KHz buzzer.

The hardware specification also incorporates the semantic mapping between actuation and sensing devices, and high-level expressions in natural language-like

ways that refer to these objects. The same object can be in fact identified by its unique label or through an attribute that unambiguously distinguishes it from other similar devices.

Considering the hardware specifications provided above, the expressions `led0` and `green led` are equivalent as they refer to the same device. The former expression identifies the object "by name", while the latter provides a natural way to refer to the same device based on an attribute, that of emitting the green light. Similarly, `sensor1` is equivalent to `light sensor`.

This equivalence is declared through the following rules that uses the predicate `hw_mapping(?DeviceID,?Semantic_Mapping)`:

```
hw_mapping(Device, [Prop,Class]):-
instance(Device,Class),
aggregate_all(count, instance(_,Class), Count),
Count>1, attribute(Device,Prop), Class\=sensor.

hw_mapping(Device, [Class]):- instance(Device,Class),
aggregate_all(count, instance(_,Class), Count),
Count=<1.

hw_mapping(Device, [Prop, Physical, Class]):-
sense(Device,Physical),
aggregate_all(count, sense(_,Physical),Count),
Count>1, attribute(Device,Prop), Class=sensor.

hw_mapping(Device, [Physical, Class]):-
sense(Device,Physical),
aggregate_all(count, sense(_,Physical), Count),
Count=<1, Class=sensor.
```

Reasonably, the above rules incorporate common-sense ways of referring to objects. In fact, when more instances of the same object class are present, an attribute is used as an object qualifier (e.g. `green led` vs `yellow led`). Otherwise,

the attribute is unnecessary as the instance can be uniquely referred to by simply using the name of its class.

The actuation effects on hardware are declared through the `hw_effect(Object, Action, HWDrivingCode, VerificationCode)` predicate. This way, code directly operating on the hardware to reach the effect (`HWDrivingCode`) or verify its outcomes (`VerificationCode`) is bound to an action expressed as shown in Section 2.2.1. In both cases code is defined as a list of executable symbols according to the model introduced in Section 2.1.

As an example, a subset of the hardware effect rules for the IRIS mote hardware, which has been used as target platform in Section 3.6, is:

```
hw_effect(led0, action(_,[led, _,on]),
[[2,porta,low],[porta, @, 2,and,0,<>]]).
hw_effect(led0, action(_,[led,_,off]),
[[2,porta,high],[porta, @, 2,and,0,=]]).
hw_effect(led1, action(_,[led, _,on]),
[[4,porta,low],[porta, @, 4,and,0,<>]]).
hw_effect(led1, action(_,[led,_,off]),
[[4,porta,high],[porta, @, 4,and,0,=]]).
hw_effect(sensor1, perception(_,[_, _]),
[[32,46, high, 32, 46, pin_output, 1, 40, pin_input,
1, 40, low, +adc, 1, adc@, -adc,
32, 46, low, 32, 46, pin_input],
[ADCL, c@, ADCH, c@, 8, lshift, or, =]]).
```

For instance, the first rule specifies that in order to change the `led0` state to `on`, the pin 1 of `porta` must be switched to the low state (`low`). The number `2` is the bitmask required to turn this pin off.

Code `porta @ 2 and 0 <>` verifies the effect of the action by fetching the value of `porta` and performing a logic and operation with the bitmask. If the value is different from `0`, then the pin is low and the `led0` is on.

For perception actions the command to read the value enables the Analog to Digital Converter (ADC) (`+adc`), fetches its value to the TOS (`adc@`), then disables

the ADC (`-adc`). In this case, the verification code fetches the value of the ADC on the TOS and checks that it equals the value that the previous execution of the command to be tested pushed below the TOS. A summary of low-level executable words used in the hardware effect specifications above is provided in Table 2.1.

The use of verification code in the oracle generation is detailed in Section 2.2.4, while the use of the hardware driving code to generate stress tests is discussed in Section 3.6.

2.2.3 Natural Language Programming Patterns for CPS

The proposed approach for CPS application development based on high-level programming patterns tries to preserve the syntactic and semantics properties of natural language. In this section, its application to sensing and actuation tasks is discussed.

Considering the CPS described in Section 2.2.2 a subset of correct actuation sentences is summarized by the following grammar:

```
Actuation -> Qualifier Actuator Action |
         Actuator Action |
         Device Action |
         turn Qualifier Actuator State |
         turn Actuator State |
         turn Device State
Action -> State | invert state |
         toggle state |
         change state to State
State -> on | off
```

These productions define programming patterns but are not sufficient to build valid rules for the CPS. Recalling that the knowledge base is tripartite, the grammar must in fact be completed with rules including Cyber-Physical concept:

```
Actuator -> lamp| buzzer | led |
Qualifier -> green | yellow |
              4KHz | 5KHz
```

as well as instances of hardware devices:

```
Device -> led0 | lamp1 | buzzer1
```

Table 2.1: Summary table of a subset of low-level words used by specification rules in Section 2.2.2. Standard Forth words are described in the upper part of the table, hardware-dependent words in the lower.

Word	Description
@	Fetch a memory word from the address left on TOS
c@	Fetches an 8-bit value from the address left on TOS
!	Store the second item on the stack to the address in TOS
+!	Increment the content of the address on TOS by the second item on TOS
<>	Leave true on the stack if the two topmost items on stack are different
=	Leave true on the stack if the two topmost items on stack are equal
abs	Compute the absolute value of the number in TOS
and	Logical and between the two topmost item on the stack. Result is written to TOS
do	Start a defined iteration. The upper and lower bound must be on the stack
I	Push the current value of the iteration counter on the stack
if	If the value in TOS is true execute the following words until the word then is reached, otherwise skip past them
+loop	Increment the current value of iteration by the number inTOS
lshift	Using the two topmost values (n, l) on the stack perform logical left bit shift of l bit-places on n leaving the result in TOS
or	Logical and between the two topmost item on the stack. Result is pushed on TOS
s"	Parse the next sequence of chars until " is encountered. The string address and its length are left on the stack
swap	Swap the two top elements of the stack
then	Conclude the selection construct and continue execution
ADCL	Leave the address of the IrisMote ADCL data register (low byte) in TOS
ADCH	Leave the address of the IrisMote ADCH data register (high byte) in TOS
+adc	Enable the ADC
adc@	Fetch the content of ADC writing it to TOS
-adc	Disable the ADC
low	Turn a port pin off
high	Turn a port pin on
porta	Leave the address of target hardware PORTA in TOS
pin_input	Set a port Data Direction Register (DDR) pin as input
pin_output	Set a port Data Direction Register (DDR) pin as output

as ground terms.

The `Action` production uses the same syntactic elements defined for actuation actions in Section 2.2.1. For instance, the program composed by the sequence of

words `turn green led on` adheres to the actuation pattern, as well as the program `4KHz buzzer on`. These programs, although syntactically and semantically correct, are valid only if the world model includes instances of these objects. For instance, the sentence `turn green led on` is not valid if the structural model of the CPS system lacks an instance of an LED that can emit green light. Analogously, the sentence `4KHz led on` is syntactically valid but it lacks semantics.

This is not unexpected as a grammar-based approach requires additional artifacts to maintain syntax and semantics consistent Guo (2016).

Hence, our rule-based system also includes software specifications to let semantics drive the syntax and to verify that software components are anchored to the structural configuration of the CPS. This also avoids complicating the oracle generation process with different specification formalisms, models and languages.

In order to bind the items used to describe the CPS system –e.g. objects, properties, and actions– to natural language grammatical categories, grammatical classes are assigned to the symbols used for action specification. For example, an actuation sentence (see Section 2.2.1) always begins with a verb. Symbols referring to objects are identified as nouns, while device attributes are qualifiers. Actuation patterns semantically link the action performed on objects (see Section 2.2.1) to high-level sentences that actually perform the action.

The simplest actuation pattern directly associates an object to its final state. The object can be referred to by its label or from a high-level expression that uniquely identifies it. The pattern is satisfied if there is an instance of that object in the Cyber-Physical domain specification.

Hence, the sequences of domain specific words: `green led on` and `led0 on` are equally valid and represent high-level sentences to make the green LED reach the `on` state. Sentences like the aforementioned `4KHz led on` does not meet the pattern specification as an LED with attribute `4KHz` does not exist in the CPS description.

More complex patterns, which refer to more structured sentences –e.g. to change the state of an object– require further syntactic specification to distinguish between transitive and phrasal verbs whose behavior in natural language is different. Transitive actions are sentences characterized by a verb, which is always the first element of the list (see actuation actions in Section 2.2.1) followed by a noun.

In such a case, the pattern format consists of an object followed by a possible action for it. The following sentences:

```
green led invert state
led0 invert state
green led change state to on
led0 change state to on
```

adhere to the actuation pattern for transitive actions.

Finally, the sentence format of the actions starting with phrasal verbs requires the verb to be followed by the object and the rest of the syntactic items of the action. Sentences such as:

```
turn green led on
turn led0 off
```

are examples of the latter.

Patterns for sensing are defined too. As in the case of actuation, a subset of the productions concerning sensing is composed of Cyber-Physical rules:

```
Physical-quantity -> light | humidity
Sensor ->  Physical-quantity sensor
```

the natural language patterns:

```
Sensing -> Physical-quantity |
           Sensor Noun |
           Device Noun |
           Action

Action -> query Device |
          query Sensor |
          read Device value |
          read Sensor value |
          get Device value|
          get Sensor value

Noun -> sample | reading | value
```

and hardware specifications:

```
Device -> sensor1 | sensor2
```

Recurring patterns of sensing actions involve only sensor objects. The simplest sentence to measure a physical quantity consists of name of the physical quantity itself, as follows:

```
light
humidity
```

As sensors measuring these physical quantities are present in the Cyber-Physical domain description, these sentences meet the pattern specification.

Another pattern for sensing sentences relates to perception actions that are simply identified by a noun representing the result of sensing, such as `sample`, `reading`, or `value`. Valid high-level sentences require the noun be preceded by the sensor object. An object is either the label associated to the sensor or the high-level expression for that device. A set of syntactically and semantically correct sentences includes:

```
light sensor reading
sensor1 value
```

Finally, a further valid syntactic structure consists of a verb followed by a sensor object and the rest of the syntactic elements composing the perception action. These sentences:

```
read sensor1
read light sensor
get sensor1 value
get light sensor value
```

are all correct sentences expressed in natural language-like fashion. Listing 2.1 reports the specification of sensing and actuation patterns using the predicate `pattern(?Pattern_type, ?Task_type, ?Pattern)`.

Listing 2.1: Rule-based specification of sensing and actuation patterns.

```
pattern(actuation,action([FinalState],[Class,_,FinalState]),[
    Object,FinalState]):- action([FinalState],[Class,_,FinalState
    ]), mapping(Device,O,Object), instance(Device,Class), Class\=
     sensor.

pattern(actuation, action(Action, [Class,_,_]), [Object,Action])
    :- action(Action, [Class,_,_]), transitive(Action),mapping(
    Device,O,Object), instance(Device,Class), Class\= sensor.

pattern(actuation, action([Verb|ActionRest], [Class,_,_]), [Verb
    , Object,ActionRest]):- action([Verb|ActionRest], [Class,_,_
    ]), phrasal(Verb),mapping(Device,O,Object), instance(Device,
    Class), Class\= sensor.

pattern(sensing, perception([Physical],[Sensor, _]),[Physical])
    :- perception([Physical],[Sensor, _]), physical_quantity(
    Physical).

pattern(sensing, perception([Physical,Noun], [Sensor,_]),[Object
    ,Noun]) :- perception([Noun], [Sensor,_]), noun(Noun),
    mapping(Sensor,O,Object).

pattern(sensing, perception([Verb|Rest],[Device,_]),[Verb,Object
    ,Rest]):- perception([Verb|Rest], [Device,_]), verb(Verb),
    mapping(Device,O,Object), instance(Device,sensor).
```

2.2.4 Automated Oracle Generation

As described in Section 2.1, the implementation of the symbolic computation paradigm fosters the possibility of developing a DSL that directly drives the hardware without the need of multiple models and languages. As this approach naturally supports the development of programs as a sequence of high-level words, a set of programming patterns have been identified whose syntax and semantics are very close to natural language sentences.

Even if the SUT conforms to one of the identified patterns it is not sure that a

behavior for the target system either exists or yelds the expected results. In fact, the implementation of each word on the target machine is left to the programmer that defines the computation associated with the execution of each symbol.

For instance, there are several possible implementations for each of the words composing the sentence `lamp on`, although having the same hardware effect. Considering the implementation introduced in Section 2.1, running the word `lamp` implies the execution of the sentence `no relay`. The word `no` may be defined to set a boolean flag that is read and put on the TOS. The word `relay` may check if the normally open relay address is contained in the TOS and if so, it pushes the address of the code to switch it on. Otherwise, it pushes on the stack the address of the code to switch the normally-close relay. Finally the word `on` may execute the code at the address which is on TOS.

However, another implementation option is that the word `relay` parses the next symbol in the text input stream to the interpreter. If the symbol `on` is encountered –that is, the comparison with the constant string `on` left on the stack equals to true– the relay is switched to close the circuit, otherwise the relay breaks it.

This example shows that as different implementations are possible for the same high-level sentence, it is useful to incorporate mechanisms to verify that the program works correctly, –i.e the lamp has reached the `on` state– during its execution.

The proposed system specifies the mapping between high-level object states and the corresponding internal hardware configuration, as described in Section 2.2.2. Hence, the system generates the oracle as the sequence of words to check that the actual hardware configuration meets the specified one. The oracle code is then embedded into the SUT while runtime verification occurs on the target platform.

The oracle generation process is schematically shown in Figure 2.5.

Given a high-level sentence as the SUT, the first step is syntactic and semantic verification to check that the sequence of words satisfies one of the specified patterns. If pattern matching fails, then the system will signal an inconsistency for that SUT. In this case there is no oracle.

Otherwise, a high-level sensing or actuation sentence has been recognized. The system then checks the existence of a specific mapping between the high-level sentence and the hardware configuration. A failure at this point signals an inconsistency between the action, which is implemented by the sentence, and the

expected high-level and hardware states. As a consequence, the oracle code is not generated. Otherwise, the oracle is generated and appended to the SUT code to be run on the target hardware.

The oracle generation does not involve any analysis of the word definitions designed by the programmer. Rather, the output code (SUT + oracle) is executed on the Forth environment of the target machine and the result is placed on the parameter stack (see chapter 4 and Appendix). A positive outcome indicates that the actual hardware configuration matches the specifications and suggests correct word implementations. For instance, given:

```
green led on
```

as SUT, the system would generate the following code:

```
green led on  porta @ 2 and 0 <>
```

As already introduced in Section 2.2.2, code `green led on` performs the action of turning the green LED on while code `porta @ 2 and 0 <>` verifies that the effect of this action is correct. As the green LED is on if pin 1 of `porta` is in the low state (`low`), the value of `porta` is read and a logic AND operation is performed with the bitmask. If the value is different from `0`, then the pin is low and the green LED is on.

At the end of the runtime execution the oracle would thus leave a truth value expressing the outcome of the test in the TOS.

The oracle generator code uses the predicate `oracle(+Sentence, -Oracle)` which is defined as follows:

```
oracle(Sentence, Oracle):-
 setof(_, pattern(_,Action,WordSubList),_),
 member(Object,WordSubList),
 mapping(Device,Class,Object),
 hw_effect(Device,Action,[_,Oracle]),
 flatten(WordSubList,Sentence).

oracle(Sentence, Oracle):-
```

```
setof(_, pattern(sensing,Action, [Physical]),_),
sense(Device,Physical),
hw_effect(Device,Action,[_,Oracle]),
flatten(Physical,Sentence).
```

The input list (`Sentence`) must meet the pattern. The high-level action in the input sentence, which has been recognized by the pattern, must match the respective hardware effect so that the corresponding oracle could be generated. The object in the high-level action must be mapped to a hardware device. In essence, the oracle generation takes place by unification with the action and the object.

The first rule associates the oracle to the input sequence adhering to all the presented patterns, except those matching the sensing pattern that are generated by the second rule (see Section 2.2.3) . The pattern, however, can produce nested lists, so it has to be flattened. For instance, provided the list of words `[green, led, on]` as input, the pattern for this actuation action will associate the output list `[[green, led], on]`, keeping separate the sequence of words defining the object from the word representing the action. The flatten predicate eliminates the divide producing a sentence that matches the pattern (`[green, led, on]`).

2.2.5 Automated Test Case Generation

The proposed system exploits Cyber-Physical rules as well as hardware and software specifications for automated test case generation. The flowchart of the automated test case synthesis is shown in Figure 2.6.

Provided with the source code as a collection of word definitions, which have been designed by the programmer, the system requires just the application domain word set as input. The first step thus just extracts only the names of each defined word from the source code.

As a result of the unification, the system generates a collection of sentences from all the possible word sequences. For each of these sequences, the system verifies that both syntactic and semantic specifications are met. Hence, valid test cases are sequences matching natural language-like patterns in the knowledge base. Each test case is provided as input to the oracle generator, as detailed in the previous

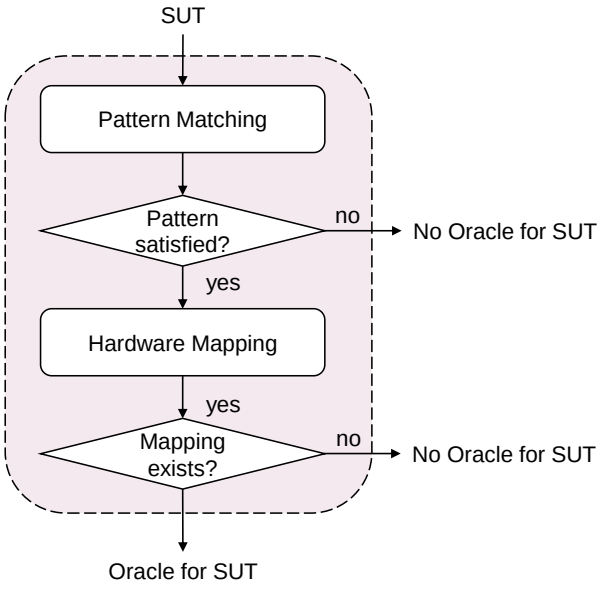


Figure 2.5: The oracle generation process.

Section. If the oracle is not generated, the sequence is removed from output test cases. Otherwise, the system incorporates the oracle into the test case. Test cases and the respective oracles, are stored in separate source files and executed one at a time for runtime verification on the target hardware.

For instance, supposing that the programmer defined the domain-specific words `green, yellow, led0, led, humidity` and `on`, the system generates the test cases reported in Figure 2.7.

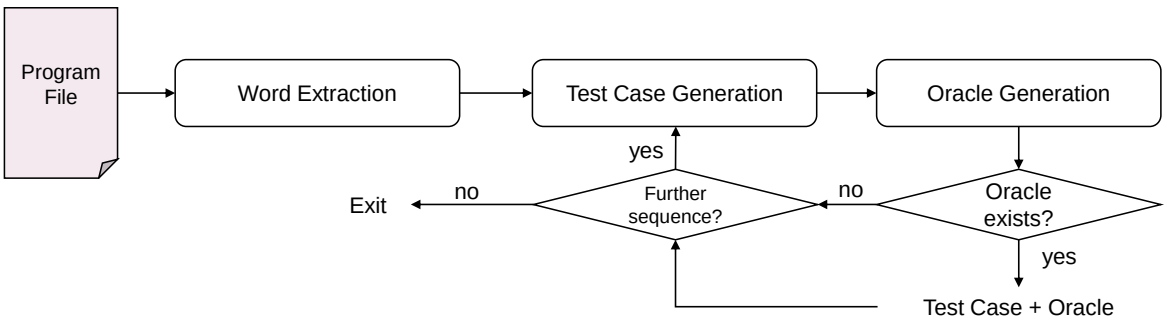


Figure 2.6: Test case generation. Given the word set defined by the programmer, the system generates all possible valid sequences along with the respective oracles.

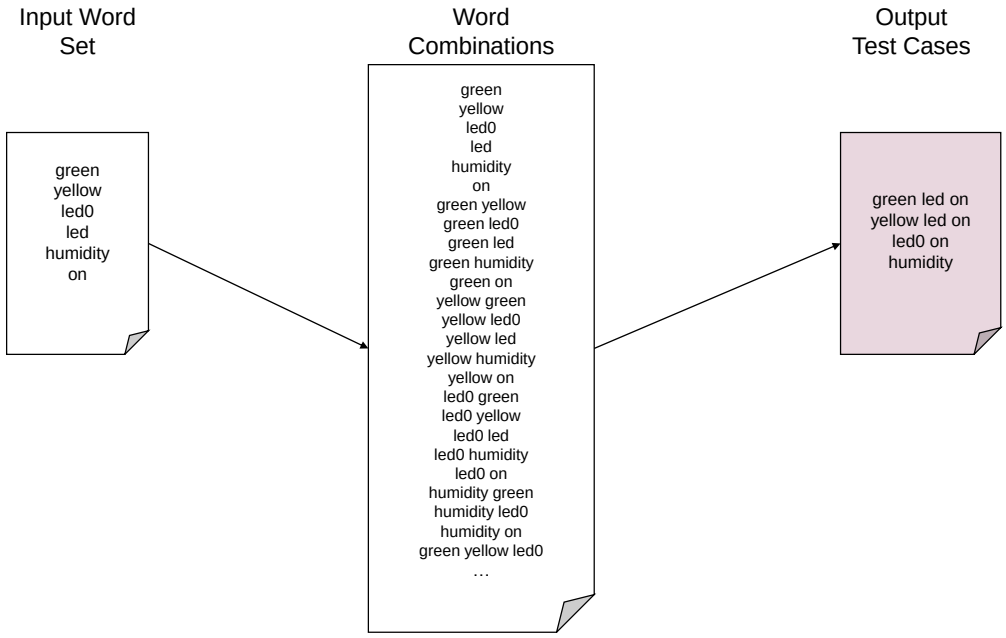


Figure 2.7: Given an input word set the system generates the test cases by combining the input words into word sequences that adhere to pattern specifications.

The automated generation of natural language-like sentences as input test cases, along with their respective oracles, uses the predicate `test(+DefinitionList, -Test, ?Oracle)` that is coded as such:

```
test(Words, Test, Oracle) :- combination(Words, Test),
       setof(A, pattern(_,_,B),_), flatten(B, Test),
       oracle(Test, Oracle).
```

A subset of all the predicates used by specifications rules provided in Section 2.2 is summarized in Table 2.2.

2.3 Stress Tests and Experimental Results

The rule-based system can be used to generate stress tests to be run on the target hardware. To this purpose, a natural language-like sentence is given as a input to the system and an oracle is generated, as described in Section 2.2.4, and appended to the sentence. The resulting code (SUT + oracle) is put in a loop that executes it

Table 2.2: Summary table of a subset of predicates used by specification rules provided in Section 2.2.

Predicate	Description
aggregate_all(+Template, :Goal, -Result))	Collect solutions of all the Template that satisfy the goal Goal and aggregate bindings in Goal according to Template
combination(+InputList,-Combination)	Generate all the possible combinations of the elements in InputList, including subsets and permutations.
flatten(+NestedList, -FlatList)	True if FlatList is a non-nested version of NestedList.
mapping(?Device, ?Expr, ?Object)	True if Object is a Device or the high-level expression Expr for Device.
setof(+Template, +Goal, -Set)	Set is a List of all the Template that satisfy the goal Goal without redundancy.

R times along with code updating statistics concerning correctness with respect to the effects on the hardware and execution time. For the correctness assessment the count of failed verifications is recorded. This loop is contained in another one that makes it execute for values of R ranging from 1 to N with a step increment of K, where N is a parameter set by the tester. For each execution of the internal loop the statistics are collected and associated with the value of R. The collected data allow to verify the correctness of the high-level code, highlighting issues triggered by increasing repetitions of the sentence like stack overflows, memory leakages, unresponsive subsystems, or conflicts with interrupt service routines. Estimates of the asymptotic trend of the execution time can also be easily plotted.

An execution time baseline estimator can also be generated that exploits the hardware specifications replacing the high-level sentence with the related sequence of low-level symbols defined in the knowledge base (see `HWDrivingCode` in Section 2.2.2). Provided that this code has been crafted as efficient as possible, driving the hardware through low-level words, the estimator provides the lower bound for the execution time, as application code is destined to trade expressiveness and abstraction for execution time.

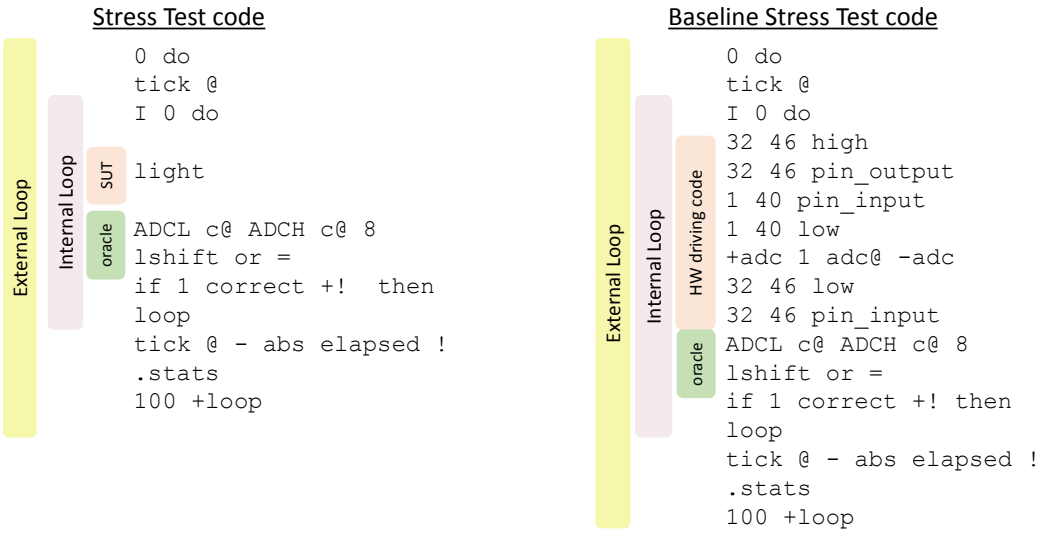


Figure 2.8: Stress test code for a high-level sentence to acquire a light sample. The high-level HW-driving code and its oracle is put in the internal loop and statistics are updated. The external loop increments the upper limit of the inner loop by 100. The word `tick` acquires the current value of the timer. The timer value is fetched at the start and at the end of the internal loop to obtain the execution time by difference.

The system has been tested with a high-level sentence to acquire a light sample as input.

The simplest form of the sentence according to the CPS patterns (see 2.2.1) is:

```
light
```

The generated stress test code is provided in Figure 2.8. The stress test code requires the the upper bound to be on the stack before execution. The step increment is 100.

The sensing operation leaves the sensory reading in the TOS. As the light sensor is connected to the ADC, the oracle is the word sequence to read the ADC register and to compare its content with the TOS. The target hardware was an IRIS mote Wireless Sensor Network node running the AmForth v. 5.4 (M.Trute, 2016) Forth environment.

Results of the stress test with the upper limit of 2000 repetitions are reported in Figure 2.9. The system behaved correctly for any repetition number. As expected,

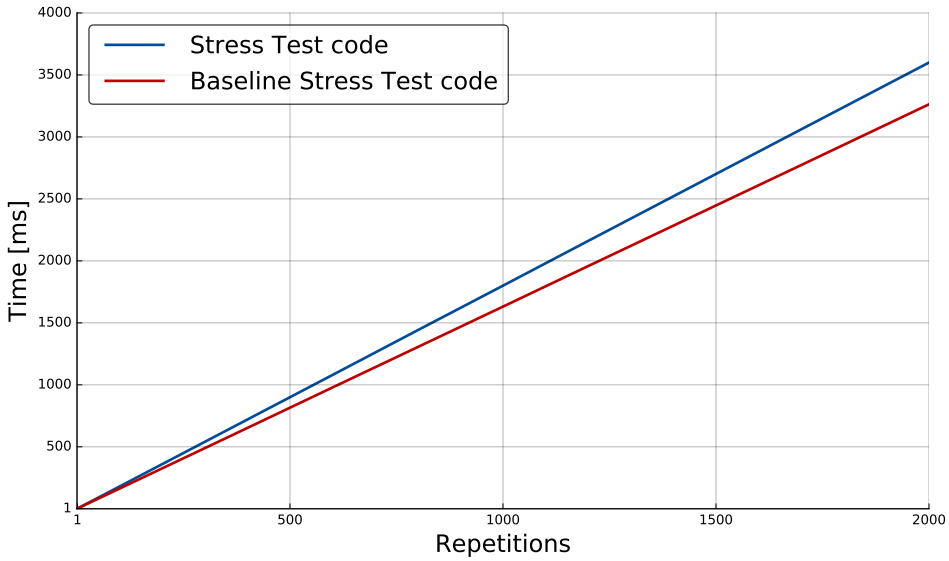


Figure 2.9: Results of the stress test

the baseline time was slightly lower than the execution time on the target platform. This difference, amounting to a few CPU cycles, is easily ascribed to the indirection code included in the definition of `light`, which is missing in the inlined definition of the Baseline stress test code. As seen in the graph plot, this difference would be actually measurable with rather low precision on the target hardware, due to the relatively large granularity of the timescale of the timer circuit with respect to the execution time of the indirection. Higher precision estimates of the execution time of the SUT may be thus obtained by increasing the number of repetitions of the test.

2.4 Discussion

To model high-level CPS application domains, a formal approach based on the integration of Cyber-Physical concepts and properties, hardware components and programming patterns has been presented.

The CPS is not modeled from a holistic point of view, but rather CPS domain is explored from a node-level perspective as the physical world has those features the cyber devices are able to sense. For instance, the world may be characterized

by a physical property, e.g. temperature, because there is a device able to measure it.

The proposed system has been used as a verification tool of resource-constrained devices as it automatically generates the oracle for a certain SUT. Although the oracle generation is offline, verification is undertaken on-board the target platform, thus making verification not partitionable according to classical criteria. The main goal is not to assess the correctness of a CPS behavior as a whole. Rather, the rule-based system is intended for the verification of independent components that can be integrated/scaled into larger systems.

Considering mainstream adopted computational paradigms, centralization often results an obvious forced choice. Symbolic computation based on high-level words allows to reach a twofold objective: (i) interoperability among different devices (ii) reduction of intermediate abstraction layers to map an abstract model to executable code. Generic ways for composing different words related to CPS domain have been specified in ways that a program can exhibit expressiveness by preserving natural language syntactic and semantics properties. Actually, according to specifications, the proposed methodology allows to define a sort of Domain-Specific Language (DSL), but verifiable on individual CPS devices. However, it is inappropriarte to speak about a DSL in a classical sense and with clearly defined boundaries as the executable language is extensible.

Interoperability arises from the definition of such a DSL, which can be used by resource-rich nodes as well as by those characterized by limited resources. Furthermore, ontologies are not needed to provide homogeneous knowledge representation. Moreover, ontologies cannot be provided to resource-constrained nodes to incorporate high-level domain knowledge. Rather, symbols themselves represent domain concepts, since words can be directly taken from the natural language dictionary and combined with each other according to patterns inspired to common language. However, the adopted approach allows to solve natural language ambiguity operatively, i.e. through software verification. Obviously, as the domain and the CPS system becomes larger and more complex, it is necessary to extend Prolog specifications about new domain concepts, hardware components, programming patterns.

Chapter 3

High-level Executable Specifications for Wireless Sensor Networks Design and Verification

Translating hardware and software specifications into a high-level programming language implementation is notably error prone and CPSs make no exception (Estevez and Marcos, 2012).

Moreover, data-sheets and specification documentation may leave out some information or provide details about typical operation cases regardless of the target operating environment, e.g. timing constraints or resources. The incorporation of runtime verification techniques is thus quite desirable to enrich information provided by data-sheets and to ensure that the system adheres to its functional specification during execution (Fischmeister and Lam, 2010).

The burden posed to the design of CPSs becomes relevant when systems include cooperative units composing Wireless Sensor Networks (WSNs) that are intrinsically characterized by severe hardware limitations (Martorella et al., 2014; Iyenghar et al., 2013).

In WSNs keeping the semantic content of the target code high is difficult, especially in the context of the interpretation of collected data. Current solutions involve the use of ontologies that allow a high level knowledge representation while maintaining tight bonds with the domain of interest (Serrano et al., 2007).

This chapter discusses how to cope with these issues using a methodology and a software platform for resource-constrained Wireless Sensor Network nodes that ease the development of abstract models on the target hardware and permit their effective execution. The semantic model of the symbolic platform, which will be presented in Chapter 4, running directly on real hardware is described, and guidelines to designers are identified in order to develop meaningful and understandable implementations even on resource constrained embedded devices.

This chapter aims at:

- Trying to bridge the gap between natural language specifications and code implementation by providing a semantically defined environment that permits high level descriptions, but executable even by resource-constrained motes;
- Identifying guidelines and key steps to incorporate semantics into the practical development through the proposed approach given an initial specification.

Designers could thus benefit from the possibility of developing code aligned with high level specifications in order to make implementations semantically obvious and to speed the correction of the code when bugs are detected (Moha et al., 2010).

To provide a working example, this methodology has been applied to specify the functional behavior of a radio transceiver chip. The resulting executable specification is also endowed with an oracle for runtime verification. The code is quite clear and strongly tied to the component description, and runs effectively on the target hardware.

In Section 3.1 the computational model is presented while in Section 3.2 its basic features and real implementation on WSN nodes are briefly introduced. In Section 3.3 guidelines and principles to develop high-level executable specifications are provided. Section 4.4 describes a step-by-step development of executable specifications of a radio transceiver chip. The obtained code also includes an oracle for its runtime on-chip verification. In Section 3.6, experimental results are presented while Section 3.7 discusses advantages and drawbacks of this semantic approach and possible future investigations.

3.1 Computational Model

While Chapter 2.1 describes how it is possible to model objects, actions, sensing task, programming patterns through sentences using symbols from the natural language, this Chapter details how these sentences are directly executable on target hardware devices and presents a methodology and a development environment based on a symbolic paradigm that is suitable for resource-constrained devices.

Leaving out the formal aspects, as in the concept of denotational semantics (Guo et al., 2014), the meaning of an expression is formally determined by the meaning of its sub-expressions, in this model the task is defined in terms of the words used to describe it. After all, also in the natural language the meaning of a sentence is given by the meaning of the individual words that make it up.

In the proposed symbolic-based model, words are executable. This implies that the obvious meaning of a word can be associated with a computation. This approach lends itself to be "compositional" since each symbol has a meaning, and the sequence of words defines the semantics of the entire computation. Therefore, this semantic model is an abstraction that maps symbols, which can be indistinctly numbers, adjectives, objects or actions, to the real world. Moreover, the interpretation of the meaning from the instances themselves permits to express information without the need for conventional meta-modeling techniques (Vyatkin, 2013) since it is based on natural language words whose meaning is obvious. In this sense, making the implementation semantically clear but executable at the same time is possible. Therefore, the code becomes similar to a high-level executable description.

In this perspective, Forth (Pelc, 2011) (see chapter 4 and Appendix) was been adopted as a foundational tool. Forth defines a methodology based on a stack machine and a stack-based programming language that does not have a formally defined syntax, but rather permits the syntax itself to be semantically defined by the order in which the words follow each other. In a Forth-based environment, new words can be easily created and added to the system *dictionary*. A word definition entails the association of a computation that is expressed as a sequence of executable words already in the dictionary. This concept is similar to the action of explaining the meaning of a word in everyday speech as a sequence of other

words of which the meaning is already known. This means that the programming paradigm is model-oriented, in the sense that the designer includes new words in the environment and this results in defining a new language made up of words that are aligned with the requirements and with the high-level domain. Therefore, this approach reduces the effort in understanding the functional behavior of the system underlying the final code during development, update or debug operations because the semantic content of the implementation code remains high in all the development stages.

In order to clarify how semantics can be easily incorporated in a symbolic environment, the description of a generic iteration of a genetic algorithm (GA) is provided as an example. This is a representative example that goes beyond the scope of this work, but can be useful to grasp the applicability of the symbolic approach to specify not only system behaviors but also high-level algorithms. A GA consists of a series of standard steps that are executed until a good or optimal solution is found. A generic iteration consists in the evaluation of the solution, in the selection of individuals for the next generation and in the application of reproductive operators like cross-over and mutation. Therefore, in this semantic model it is possible to define a genetic algorithm step (GA-step) as follows:

```
: GA-step
  evaluation selection reproduction ;
```

Colon is the Forth word to begin a word definition, in this case the word named `GA-step`, while `;` ends a word definition. In the context of the definition of a genetic algorithm, the words `evaluation`, `selection` and `reproduction` are semantically obvious and reflects the same words used in natural language to describe it with a high level of abstraction. The word `reproduction` may have been previously defined as the sequence of the words that represents the reproduction operators `cross-over` and `mutation`:

```
: reproduction
  cross-over mutation ;
```

The overall computation is highly abstract through the description of a generic flow diagram in terms through the word `GA-flow`:

```
: GA-flow
  population initialize
  begin GA-step  solution? until ;
```

where `begin` and `until` are words already included in the environment, to define a loop running until `solution?` becomes true. The semantics of `GA-step` is associated to the computation incorporated in the word definition.

3.2 A Semantic Environment running on WSN nodes

Wireless Sensor Networks are characterized by several devices even deployed on hostile environments, often not permitting human interventions. For these reasons, having an environment running on the nodes that fosters the equality among specification and code implementation, also from a morphological point of view, may support the drafting of correct meaningful code.

Indeed, the combination of a symbolic computation with an interactive programming methodology have been investigated to make testing take place during development (Gaglio et al., 2014). Forth words for implementing sensing and actuation tasks as well as for networking abstractions have been developed to support typical WSN operations.

All of these features will be presented in Chapter 4. To specify distributed computing tasks, since in the natural description of a cooperative task it can be said that a node tells another node to do something, a syntactic construct that perfectly fits this informal description is proposed. Interactivity, symbolic processing and executable code exchange are the pivotal characteristics of the proposed software system.

According to the Forth computational paradigm, next Chapter will describe a programming environment in which it is possible to develop code structurally similar to the original specification, by encapsulating lower level implementations,– i.e. to set ports, registers and so on– in expressive words operating directly on the hardware device. This lays the basis to develop code that is fully aligned with

its functional description, but in an effective way since it runs directly on the hardware without any other intermediate stage.

While syntactically correct statements in most widespread programming languages are formally described by a grammar, the syntax of Forth is deep-rooted on semantics (Stoddart et al., 2012). Additionally, Forth is not only interpreted but offers on-board compilation as a standard programming method.

Other attempts to bring onboard interpreters to resource constrained devices target either simple languages as BASIC (Miller et al., 2009) or higher-level languages, such as Java and Python. These approaches do not offer enough expressiveness to justify their huge resource consumption, though.

3.3 Key Steps to Make High-Level Descriptions Executable

The main objective of this section is to define guidelines to attain symbolic code that is easily understandable and resembling as much as possible to a natural language description. In the best case, the final code would be similar to a sentence in natural language. The designer wold proceed according to the following steps:

- **Step 1: Understanding the functional behavior.** The primary step is grasping and abstracting the main aspects of the system. For distributed computing applications, as the proposed paradigm allows for the exchange of executable code between nodes, the task must be implemented according to an “interaction-oriented" model as a set of computations that take place locally and interactions in which a node tells another one what to do (Gaglio et al., 2016).

- **Step 2: Identifying key concepts.** After the whole high-level operation has been figured out, it is necessary to extract the key concepts from the specification expressed in natural language. A good practice is to try to explain the operation by putting the principal concepts into words. This step allows to identify the essential parts and to decompose the problem,– e.g. system operation, algorithms and distributed protocols– into smaller compu-

tations. In this sense, the general concept, that is, the whole computation, is given by the meaning of the concepts that compose it.

- **Step 3: Mapping concepts to words.** The main concepts already identified in the previous step are quite abstract and can be thought of as the words of the high level code in the semantic model. To maintain a high semantic content, is therefore good practice to include words whose name refers to the computation associated with it. As in the example in the previous section, the word `reproduction` is made up by the concepts of `cross-over` and `mutation`. In turn, `cross-over` is defined in terms of words whose sequence indicates the computation to perform the cross-over and so on. Therefore, the design proceeds through a top-down approach, from general concepts to more specialized ones. Essentially, a generic concept is the composition of more specific concepts as well as a high-level word is defined on the basis of more specific words.

- **Step 4: Defining the word set.** In this phase coding is carried-out, as words are being defined on real hardware devices. While the design process proceeds in a top-down manner, coding follows a bottom-up path. In fact, as the definition of new words is based on words that are already defined, more specific words must be coded before more general ones.

  The proposed environment is interactive even on deployed nodes through the wireless connection, and this feature makes it possible to define a word and immediately test its operation. Code correction simply involves the redefinition of the word, i.e. its association with another computation. The steps identified for the development of semantic rich code in the proposed environment are summarized in Figure 3.1.

These guiding principles are applicable even for the development of distributed protocols in the WSN scenario as well as for the development of hardware component drivers. Nevertheless, the design of new protocols and components goes beyond the aim of this discussion.

Next section focuses instead on practical development on real resource-poor nodes, and on the attainment of executable code similar to specifications for hard-

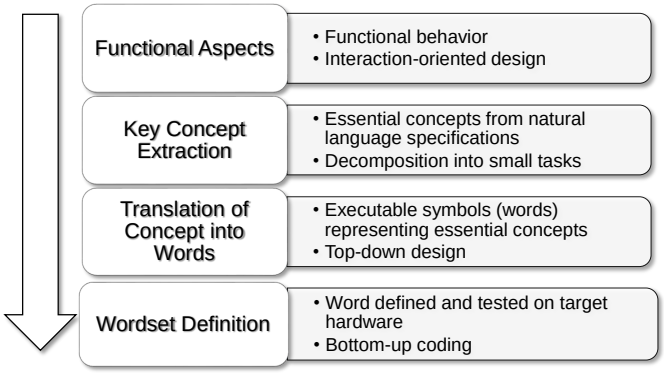


Figure 3.1: Key steps guiding the implementation of tasks aligned with their informal description. After understanding the whole operation from a functional point of view, it is necessary to decompose it into subtasks. Then it is required to identify the main concepts and translate them into high level words. More generic words are composed of more specific words. Their coding takes place following a bottom-up approach.

ware components.

3.4 Case Study: Executable Specifications for Runtime Verification of an On-Board Radio

In this section the outlined methodology is applied to show how to map informal specifications provided by data-sheets to a high-level running implementation that drives a hardware component. In particular, the AT86RF230 transceiver chip (Atmel-ATRF230, 2016) is considered, as it is embedded in several reference WSN hardware platforms.

For verification purposes, the development of an online oracle that monitors radio operations is described. The goal is not only to check that the subsystem complies with the specifications, but also to discover information omitted by the reference documentation or to enrich it.

Usually, the implementation of an oracle on resource-constrained systems involves considerable overhead (Iyenghar et al., 2013). Instead, the outlined methodology allows for an implementation with a low memory footprint.

3.4.1 Step 1: Understanding the functional behavior

According to data-sheets, the abstract model representing the radio operation is a finite state machine (FSM). Radio key functionalities, such as transmission or frame reception, are enabled by performing state transitions.

Henceforth, italics is used for some keywords in the system functional description that can be found in the final code.

Following the specification, a *state reaches* another state by *running* a certain command. This occurs either by writing the transition identification number to a predefined radio register, by asyncronous *events*, or by rising or lowering the *SLP_TR* pin.

Symbols performing state transitions are: *trx_off, pll_on, rx_on*, *force_trx_off, tx_start*, *sleep*. The *sfd_detected* event indicating the detection of an incoming valid frame and the *frame_end event* to signal the end of either reception or transmission also cause state transitions. The AT86RF230 transceiver distinguishes between six events on the same interrupt line that are *dispatched* by reading the *IRQ_STATUS* register. A *transmit* operation can be started by writing *tx_start* in the appropriate register, provided that the radio is in the *pll_on* state, while frame reception is enabled in the *rx_on* state.

A successful state change can be confirmed by reading the radio transceiver status in the *TRX_STATUS* register.

The abstract model arising from the first step is illustrated in Figure 3.2.

3.4.2 Step 2: Identifying key concepts

The names to identify states as well as events and commands, which enable state transitions, are definitely the key concepts of the high-level abstract model. However, in this step, non functional requirements, such as state transition timing, which are also provided by the reference documentation, are needed to enrich the FSM description. Therefore, another useful concept that has been identified to abstract the FSM operation of real hardware devices is the *typical time* needed to perform a state transition.

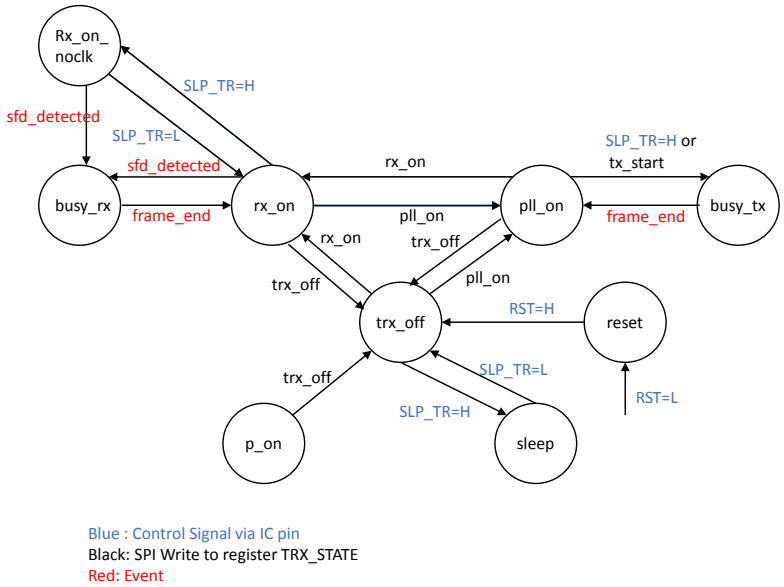


Figure 3.2: *Step 1.* The radio transceiver chip operation is described by a finite state machine in the data-sheets. The `frame_end` event in transmission is actually generated in a different way, as found with the Runtime Component Verification Tool (Sect. 3.6.2).

3.4.3 Step 3: Mapping concepts to words

Keywords should be aligned with their specific code-implementation. Proceeding from the top downwards, as indicated in Figure 3.3, high-level words are decomposed into more specific concepts. Word names should be chosen as close as possible to those used in the description. This step should not be underestimated, as it is essential to maintain the semantic content of the specification in the names of words. The top-down approach is reflected in the design of the words that proceed towards a gradually lower level of abstraction until built-in words are reached.

As an example, performing `trx_off` involves executing the command associated to the `trx_off` state. The word `cmd_wr` internally uses the word `reg_wr` that writes to the `TRX_STATUS` radio register using the SPI interface.

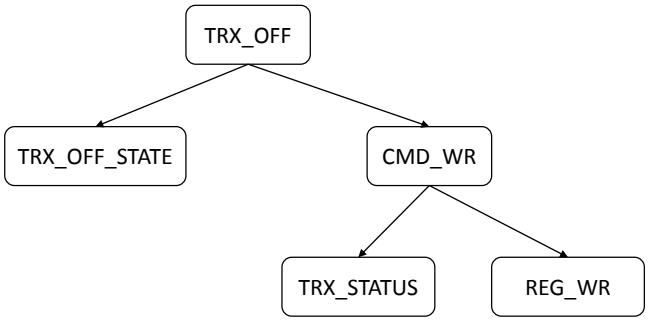


Figure 3.3: *Step 3.* High level concepts must be translated into expressive words. The design proceeds following a top-down approach, from general concepts that are progressively specialized.

3.4.4 Step 4: Defining the word set

Let us translate the high level functional abstract model in Figure 3.2 into a program to be executed on the target machine for the component verification.

The word set to specify FSMs includes words to express concepts like:

- events
- states
- symbols
- state transitions

The words `state:` and `symbol:` define states and symbols of the radio module, as follows:

```
8 state: trx_off
symbol: trx_off
```

The code is meaningful enough to grasp that the two uses of `trx_off` are semantically different. In the first case it indicates a state name, preceded by a state number, and in the latter the symbol that causes the identically named state transition.

The word `event:` is used to define high-level events, as follows:

```
event: frame_end
```

The words used to refer to symbols, states and events are rather generic and can be used to specify other hardware components whose functional behavior can be represented with a FSM.

To provide a generic and abstract way of specifying FSM state transitions, the word `running:` is used to specify an input symbol or event causing the state transition, while the word `reaches:` is used to express the transition among two states caused by the execution of the specified symbol. The underlying idea is to translate the FSM into expressive sentences that bring to mind the graphical representation. For instance, running `trx_off`, the `pll_on` state reaches the `trx_off` state. This is expressed by the following target code:

```
running: trx_off state: pll_on reaches: trx_off
```

Linking this piece of code to its graphical representation is immediate.

Once the target device executes the code, the specified state transition is stored into the internal representation of the FSM model. Moreover, to provide the possibility of specifying actions to be undertaken once a state is reached, the optional word `doing:` can be appended to the previous code snippet as follows:

```
running: trx_off state: pll_on  reaches: trx_off
doing: nothing
```

The word `doing:` must be followed by a word whose associated computation is the sequence of actions to be done reaching the arrival state. In this case, the computation associated to the word `nothing` is a no-operation.

Previous steps also highlighted the high-level concept of transition timing. As the data-sheet associates a unique symbol to each transition timing, the word `time:` is used to specify such a symbol. For instance, as found in the chip specification document, `tr5` is the symbol identifying the state transition from the `pll_on` state to `trx_off`. The typical time spent in this state transition is 1 μs. Such a transition timing is expressed by:

```
1 us time: tr5
```

To add information about the time required for a state change, if present in the reference specification document, the optional word `taking:` must be followed by the word referring to that time transition and appended to the previous code snippet:

```
running: trx_off state: pll_on  reaches: trx_off
doing: nothing  taking: tr5 time
```

This way of specifying state transitions is used with events. Therefore, according to the radio state diagram:

```
running: frame_end  state: busy_tx
reaches: pll_on
doing: nothing  taking: tr11 time
```

Finally, once a minimal wordset to describe the FSM formalism has been introduced, its applicability to real hardware components requires mapping high-level events of the FSM abstract model to low-level interrupting events.

The AT86RF230 transceiver distinguishes between six events on the same interrupt line, although only two of them signal state transitions and are reported in the state diagram. A dispatcher is thus needed to determine the cause of the interrupt.

Our symbolic environment also permits to develop a semantically clear executable description for a dispatcher component.

The following code provides a way to associate dispatching conditions with discriminable events on an interrupt line:

```
dispatcher rf230-dispatcher
 conditions: ( -- 0 )
 cond: irq_value trx_end    equals  ;cond --> frame_end
 cond: irq_value rx_start   equals  ;cond --> sfd_detected
 cond: irq_value pll_lock   equals  ;cond --> noop
 cond: irq_value pll_unlock equals  ;cond --> noop
 cond: irq_value trx_ur     equals  ;cond --> noop
 cond: irq_value bat_low    equals  ;cond --> noop
 ;conditions
```

Table 3.1: Summary table of relevant words used for the radio component abstraction and verification

Word	Action
events	Store the total number of events in the FSM in a status variable
symbols	Store the number of symbols causing state transitions in a status variable
states	Store the number of states in the FSM model in a status variable
check	Check if the radio current state matches the expected one
update-state	Update the current state after a state transition
start:	Store the initial state for simulation
sim	Reset all the status variables used for statistics
.stats	Output, either the UART or the radio, runtime monitor statistics
expected+	Increment the expected time variable
actual+	Increment the actual time variable
redefine	Perform a symbol redefinition insert an online monitoring hook
statistics!	Store the result about the last state transition
+monitor	Enable the online oracle
-monitor	Disable the online oracle

The word `dispatcher` is used to define a dispatcher named `rf230-dispatcher`. Then six conditions are enumerated in the `conditions:` block, each one is enclosed in a

`cond:` <*code*> `;cond` syntactic construct. The word `equals` compares the signaling bit of the irq register to the bitmask of a state, while `->` is used to specify the word to be executed once the condition is verified. Events are executable symbols corresponding to interrupt service routines to handle low level interrupts. For instance, a `frame_end` event is triggered once the `irq_value`, i.e. the value in the *IRQ_STATUS* register, has the `trx_end` bit set.

Event and bit names are exactly the same reported in the data-sheet.

3.5 A Runtime Component Verification Tool

The executable specification obtained by previous steps is then used for the runtime verification of hardware components whose functional model implements the FSM formalism.

Since specifications are executed on the target hardware platform, this word set has been exploited to easily include an oracle to verify that the radio transceiver adheres to its state machine model during execution.

Implementing an oracle, given the executable specification, just requires to redefine words whose execution causes a state transition. Actually, the word `running:` performs the word redefinition by prepending `preamble` and appending `conclusion` to the original computation associated with these symbols. The words `preamble` and `conclusion` are *deferred words* (see Appendix), as their computational behavior can be changed at runtime. For instance, the word `trx_off` is redefined as follows:

```
: trx_off   preamble trx_off conclusion ;
```

As long as `preamble` and `conclusion` are set to a `noop` operation, the online monitoring feature is disabled.

In the proposed implementation, the monitor uses status variables storing the expected radio state as well as some statistics about the expected and actual transition timing, and the current number of right and wrong transitions done with respect to specifications. The word `preamble` checks that the current radio state, which is accessed by reading the `TRX_STATUS` radio register, and the content of the status variable are equal. The word `conclusion` accesses the timer, and enters the FSM table to get the expected reached state, and the corresponding transition timing, according to the specification provided by the following syntactic construct:

```
running: <Symbol> state: <State1>
reaches: <State2> taking: <T> time
```

Then, the actual radio state is compared to the expected one and statistics are updated, including the actual time needed for state transitions.

Finally, in order to ask the oracle if a task can be carried out within a certain time window or the minimum time it is required to perform it, two further words have been defined, `window` and `at-least`.

The idea underlying the online monitor implementation is to provide a tool to speed up the verification of embedded system component prior to deployment.

For instance, the experimental evaluation, which is detailed in the next section, shows that actual transition timings are always lower than typical timings provided by reference documents. Beyond step-by-step verification, the oracle is used to complement the information provided by data-sheets and to tailor them to target hardware and software platforms.

3.6 Experimental Evaluation

This section discusses, through several tests, the use of the runtime verification tool to gather useful metrics by running the specification on the target platform. In all but the first test, symbolic code is exchanged by nodes and locally executed to perform the respective task.

3.6.1 FSM State Path Test

In the first test the runtime oracle monitored the radio operation of the reference platform during the execution of a task consisting in repeatedly switching off and on the on-board transceiver.

From the abstract model perspective, this high-level task involves the transition from the `pll_on` state to the `trx_off` state and back again to the `pll_on` state. The test has been performed progressively increasing the number of repetitions up to a maximum of 800. The oracle response confirmed that, during all the test executions, the radio operation met the specifications as no mismatch between expected and actual states occurred.

The runtime verification tool also provided useful information about the transition timing. Results in Figure 3.4 show that the time required to traverse the state path, which is calculated according to typical transition timing in the data-sheets, always exceeded the actual time needed for this execution on the real target machine. The overall turnaround time needed to perform 10000 repetitions was 244 seconds. However, the time due to state transitions was 454740 μs, as computed by the monitor. The remaining fraction of the turnaround time can be ascribed to the interpretation overhead due to monitor execution and parameter updates.

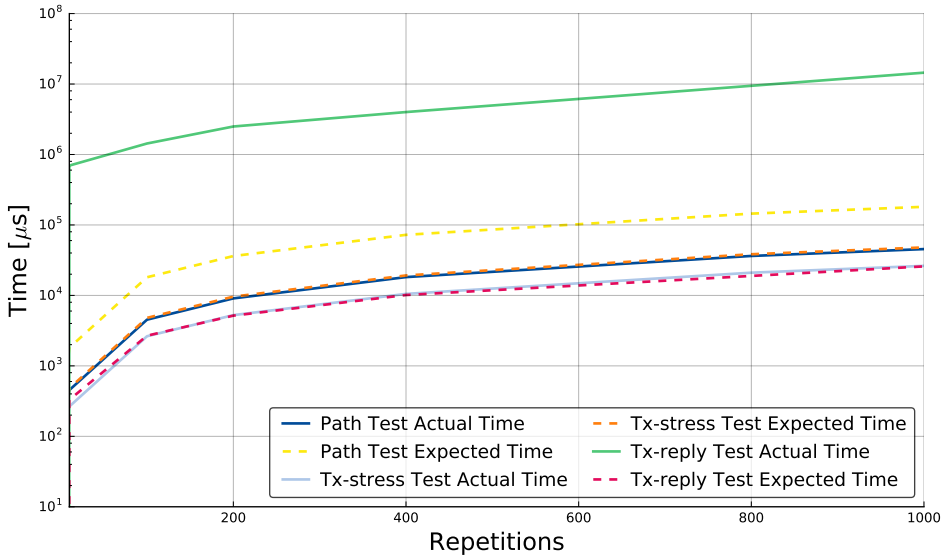


Figure 3.4: Expected and actual timings for the FSM State Path Test, Transmission Stress Test, and Transmission and Reply Test, as repetitions increase.

3.6.2 Transmission Stress Test

Whereas switching the radio transmitter on and off does not involve dealing with interrupts, the correct transmission of a frame is signaled by a high-level radio event.

The aim of the transmission stress test was to analyze the radio behavior during repeated frame transmissions in order to verify that events are handled according to specifications.

A 26-byte IEEE802.15.4-2003 standard compliant frame with a payload consisting in the code to turn the green LED of a deployed node on was used. The data collected by the oracle shows that, even in this case, the actual time spent to perform repeated frame transmissions was lower than the value reported in the data-sheet, as reported in Figure 3.4.

However, the oracle indicated a violation of the specifications, revealing incorrect state transitions although frame transmission proceeded properly. This seeming incoherence occurred when the oracle found the radio in the `pll_on` state whereas it expected it to be in the `busy_tx` state on the `frame_end` in transmission (Figure 3.2). In fact, only having a look at the FSM diagram included in

the transceiver specification document, the asynchronous `frame_end` event seemed responsible of the state change from `busy_tx` to `pll_on`.

Instead, in another part of the documentation is reported that after a correct transmission the system re-enters the `pll_on` state, and only after that the `frame_end` event is generated. To resolve such a discrepancy, as interrupting events are asynchronous and can be triggered by the system at any time, a refinement process of the executable specification was needed that permitted treating transitions caused by events and symbols differently. In fact, events signal that a state has been reached. Therefore, the executable specifications have been slightly changed to take into account events by using definitions like:

```
occurring: frame_end reached: pll_on
from: busy_tx taking: tr11 time
```

This way, a `frame_end` event detection in the `pll_on` state made the oracle consider state transitions as correct, even if it was unable to assess the radio had transitioned through the operatively "invisible" `busy_tx` state. The code for model abstraction and hardware component verification required less than 200 bytes of RAM and 2 KB of Flash memory on IRISmote. A summary of the word set, excluding already discussed words, is reported in Table 3.1.

3.6.3 Transmission and Reply Test

The transmission and reply test was aimed at assessing the behavior of the radio component in a distributed computation task. The experimental setup consists of two WSN nodes. In turn, each node incremented a shared value, stored it and then immediately sent it to the other node along with the sequence of symbols to command it to do the same. The exchange continued until the maximum number of repetitions was reached on a node. This test made intensive use of the radio so the symbolic interrupt management code was particularly stressed. The time needed to perform the task for increasing numbers of repetitions is shown in Figure 3.4. In this case, the actual time exceeded the expected one as the monitor measures, besides the time needed for state transitions, the time the radio transceiver remained in the reception state until the event signaling reception

occurred. Actually, application dependent time can be estimated by difference once the code has been verified to meet the specifications. Similar measurements are often used as metrics to assess energy consumption. In this case, the monitor provided precise results in a real application on real hardware, without needing to resort to simulated environments.

3.7 Discussion

Programming resource constrained devices with semantically clear code supports the development of correct distributed applications by reducing the semantic gap between implementation and informal description. Once bugs and errors occur, interactive programming permits to test the code during its refinement while maintaining strict ties to the related specifications expressed in natural language. Indeed, code retains a close resemblance to the functional description of the system. This makes it simple translating the specifications into executable code and the code back again into specifications. To make a task implementation tightly coupled with its informal description, word names are required to be representative of the task they accomplish and, ideally, the same used in the high level specifications. Another question concerns their order, which must be as similar as possible to sentences in natural language like those found in the specifications, but reckoning with their effects on the stack, since the execution model is stack-based. As the presented programming environment does not force to follow rigid syntactic constraints, taking into account the natural language syntax in drafting the code is not only possible but it can also be deemed a good practice.

Nevertheless, some descriptions may be incomplete or not accurate enough to describe the key steps of a task. This leads to an eventual separation between specification and implementation since it remarks the difference between *what has to be done and how it should be done*. On the contrary, providing a tool able to describe meaningfully even what is missing in the specification enhances the semantic clarity of the code. This leads to an iterative process that consists in reviewing the specification in the light of what has been discovered during the coding phase, and subsequently refining the code accordingly until the description is precise enough. However it is always possible to reach a higher level of abstraction. Encapsulating

words into other, previously defined, ones is a code refactoring process that departs from low-level details until the desired degree of expressivity has been reached. In many cases, drafting code observing the specification is immediate and does not entail many layers of abstractions. On the other hand, it is quite easy going back to the specifications.

As further examples of the flexibility of the presented approach, the proposed methodology has been applied to the development of other low-level control words.

Although the interactivity of the testing phase makes it fast, it cannot prevent from every malfunction and error. It is impossible, in fact, for the designer to enumerate all the potentially occurring situations as well as the behavior of the system for all the possible inputs. Formal specification and verification systems, as that provided in Chapter 2, help overcome these issues while allowing for automatic translation of specifications to executable code without compromising the expressivity of the final code.

Chapter 4

DC4CD: a Platform for Distributed Computing on Constrained Devices

The availability of small scale interconnected objects has given rise to promising research that considers even resource constrained devices as actors in large scale applications accomplishing complex tasks in a cooperative way (Kortuem et al., 2010).

Unfortunately, programming CPS resource-constrained devices, e.g. WSNs, is still particularly burdensome. In the common practices the application is cross-compiled and then uploaded to nodes through wired connections. However, after deployment, code updates may be needed to improve code, install new features, and fix bugs. For each update the whole programming cycle must be repeated. More importantly, depending on the target application, resource constrained nodes may be required to exhibit more advanced skills than simple sensing. Symbolic reasoning, distributed computing, high level information exchange are complex tasks that are difficult to implement due to the inner resource limitations of these devices. Therefore, novel software architectures that allow for expanding and modifying the functionalities of remote devices, natively supporting symbolic and cooperative computational models, without colliding with nodes limitations are strongly required (Xu et al., 2014).

This Chapter presents a novel software architecture, Distributed Computing for Constrained Devices (DC4CD), which is primarily designed for distributed computing and symbolic programming on resource constrained devices. The platform is based on the interpretation and distribution of code as plain text strings, rather than as bytecode. Differently from mainstream approaches, this choice does not sacrifies code compactness and provides a straightforward way to distributed symbolic processing. Moreover, it abstracts the characteristics of the target hardware enhancing the interoperability between heterogeneous devices.

Compared to similar interpretation-based architectures, the proposed implementation proves to be more efficient in terms of memory footprint and code compactness. This lays the basis for the implementation of distributed complex applications even on resource-constrained devices.

The rest of chapter is organized as follows. Related work is discussed in Section 4.1, while Section 4.2 details the computational paradigm of the proposed architecture. The platform design is described in Section 4.3. Section 4.4 presents a simplified case study for distributed measurement aggregation while Section 4.5 provides an experimental evaluation of the proposed architecture and comparisons with other similar platforms. Finally, Section 4.6 reports further discussions and chapter conclusions.

4.1 Related Work

Interpretation-based architectures for resource constrained devices have been widely proposed in literature.

Considering WSN nodes, the first virtual machine introduced for research purposes is Maté (Levis and Culler, 2002), also called Bombilla. It is a bytecode interpreter running atop TinyOS-1.x, which is an old version of the widespread WSN OS TinyOS (Levis et al., 2005b). Maté targets MSP430, MICA2 and MICAz hardware platforms and supports TinyScript, an assembly-like language. Although it is currently unsupported by more recent versions of TinyOS, it is still considered a reference platform. Similarly to the proposed architectural model, Maté is a stack-based virtual machine. The idea underlying Maté is to provide an ASVM, i.e. a VM tailored to a particular deployment, rather than a general

purpose implementation, such as DC4CD. This strongly limits its adoption for complex application scenarios as just few instructions are reserved for user customization. Moreover, a recompilation of the whole virtual machine is required to modify user-defined instructions. Code injection takes place through rigid schemes for code dissemination that affects just the application layer, while virtual machine updates require to modify the TinyOS module implementing the interpreter itself.

More recent approaches proposed virtual machines designed for well-known interpreted languages such as Java and Python. T-RES (Alessandrelli et al., 2013) and PyFUNS (Bocchino et al., 2015) are built above PyMite, a reduced Python virtual machine especially conceived for embedded systems. Their operation is restricted to RESTful architectures using the CoAP protocol. Despite these virtual machines target a resource rich platform as WiSMote, the support to IPv6 and CoAP almost saturates the available RAM, where scripts are injected and stored. This severely limits the injection of tasks more complex than simple environmental monitoring. T-RES applications are designed according to a dataflow approach by observing input or output resources of peer nodes, e.g. sensory readings or actuator values. Unlike DC4CD, T-RES does not allow for other development paradigms. In order to reprogram a node, it is possible to act just at the application level, since the platform does not permit architectural modifications of the virtual machine or the Contiki operating system. Darjeeling (Brouwers et al., 2009b) and TakaTuka (Aslam et al., 2010) are virtual machines that run just a subset of the Java language. Both these virtual machines have been ported to Contiki and TinyOS. Darjeeling applications are stored as so-called *infusion* files in Flash memory, while TakaTuka scripts are kept in RAM. Both the Java platforms target MSP430 and implement compression and compaction techniques to reduce the bytecode size.

Concerning code efficiency, an alternative solution to the injection of assembly code was implemented in the REEL framework (Alippi et al., 2011) to reduce the overhead caused by the interpretation of injected code. REEL targets Arduino and combines the compactness provided by a virtual machine instruction set with the efficiency of native-code loaders. This is achieved by decoupling interpretation and execution. Interpretation occurs just on code reception and allows for the on-board conversion of a Matlab-like script into machine code, while the execution

phase takes place frequently, but on different sets of acquired data. However, its implementation is restricted to traditional WSN tasks, such as local processing of sensed data, while support for distributed and symbolic processing is missing.

A symbolic approach on resource constrained devices had been already proposed by the authors of SensorScheme (Evers et al., 2007b), an interpretation-based platform for tiny nodes implemented above both Contiki and TinyOS. SensorScheme is similar to DC4CD as high-level Scheme code is directly injected and executed as plain text, without further intermediate translation steps. The high level *eval* primitive executes the incoming packet content. However, only application code can be injected, while modifications of both the virtual machine and the underlying OS are not allowed. Although it is still indicated as an active project, it seems no longer being developed. Indeed, poor documentation and the lack of recent updates make its installation and, consequently, comparisons with other platforms, quite difficult.

Other interpreters for known languages such as NTU-Preters (Lien and Wu, 2014), which is a BASIC interpreter, as well as for newly introduced languages such as SCript (Dunkels, 2006), which targets the MSP430 MCU, have been also proposed and implemented as applications running above Contiki. Unfortunately, the source code of such implementations is unavailable.

All in all, existing platforms are often implemented above general purpose operating systems, and, in some cases, this leads to high memory RAM requirements that collide with the node resource constraints. Moreover, interpretation-based solutions already presented in literature mainly focus on reprogramming already deployed nodes only at the application code level. Finally, the architectures proposed so far do not provide natively any networking abstraction to support distributed computing schemes.

4.2 Platform Design

The proposed software platform for resource-constrained devices implements an abstract architecture based on a symbolic computational paradigm that deals with events of the physical world, as shown in Figure 4.1.

Physical processes, as well as those related to the communication among en-

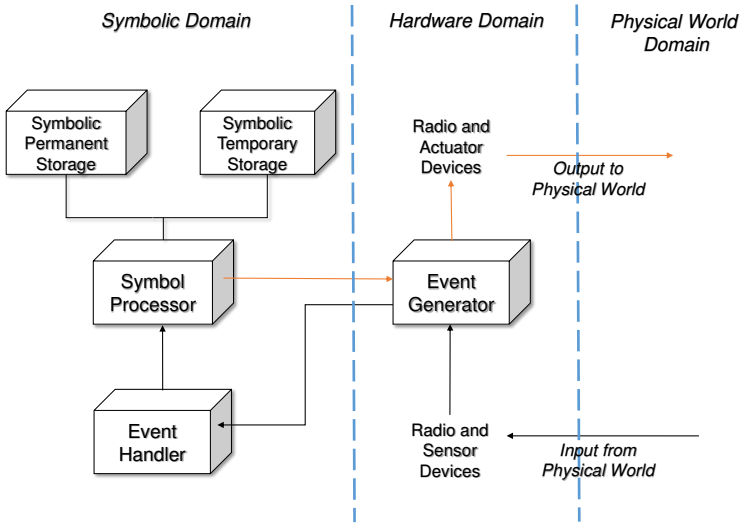


Figure 4.1: Component-based representation of the abstract symbolic machine that maps events of the physical world into high-level symbols.

tities, produce events that are primarily detected by the hardware components and mapped into a lower level representation. Events are generated and signaled through specific interfaces to the abstract component, the *Event Generator*. Beyond the simple notification of given conditions, some events may generate data or additional information. For instance, the reception of a radio message raises a hardware event that produces further data, i.e. the content of the message, which has to be processed. Notification and data events are detected by the *Event Generator* and managed by the *Event Handler* through a sequence of high-level symbols.

The hardware can thus be considered as the interface between the external physical world and the internal symbolic representation.

In this abstract architecture both data and instructions are expressed symbolically. As discussed in previous chapters, programs are merely sequences of high-level symbols called *words*. The *Symbol Processor* runs symbolic programs by sequentially evaluating such symbols, one at a time, while checking for external events. Control-flow constructs can be simply added to the model by defining symbols to mark points in the sequence that are origins and destinations of branches,

as well as symbols to jump, conditionally or unconditionally, to marked points.

The *PStg* (see Appendix) contains all the words composing the symbolic instruction set of the abstract machine. Such word set is extendible as a new symbol can be defined in terms of symbols already included in the *PStg*.

A representative example of treating a physical quantity by a sequence of high level words, is the symbolic program to acquire and store a temperature sample as follows:

```
temperature sample  store
```

A symbol can be mapped to the executable code through one of the well-known execution models for virtual machines: native, DTC, ITC (Ertl, 2001). The link between a symbol and its code is called *execution token* (*xt*). A symbol name can be retrieved knowing its associated *xt* and vice versa. This bidirectional mapping is the key to vectored execution and runtime symbol manipulations.

The abstract machine functioning relies on two stacks. The *PS* holds the instruction operands and, eventually, the operation result is placed on the stack, e.g. an instruction to store a value on a memory location requires that both the memory address and the value to be stored are the topmost items on the stack. The *RS* is used to hold return addresses for nested instructions and temporary data. Both *PS* and *RS* may hold memory addresses and integer values. The *TStg* holds the value associated with variables.

This abstract architecture is used as the basis for the platform implementation because the execution model it implements is among the simplest but it is also effective.

The proposed software platform is implemented in Forth, which naturally embodies the operation of the abstract machine described so far. Further information about Forth is provided in Appendix.

4.3 System Implementation

In the following the implementation of DC4C is detailed. Firstly, it is showed how the symbolic model is embodied in a real hardware platform. Then, this section

describes the functionalities that are currently supported by implementing a number of words grouped into word sets, as well as the communication modes provided by the proposed software platform. Finally, the abstract mechanism supporting the injection of symbolic code among already deployed nodes is detailed.

4.3.1 Implementation of the Symbolic Model

A microcontroller architecture, which is an established choice for WSN nodes, is the target core hardware to implement the abstract model introduced in Section 4.2.

An MCU typically integrates different memory units, such as Flash or EEPROM permanent memories, and volatile RAM storage.

DC4CD currently targets IRIS, which is a widely adopted resource constrained WSN hardware platform. It is based on the Atmega1281 8-bit Harvard RISC MCU running at about 8 MHz and provided with a 128 KB Program Flash memory, a 8 KB RAM, a 4 KB EEPROM, and a 512 KB Measurement Flash. Analog Inputs, Digital I/O, I2C, SPI and UART interfaces are exposed through a 51-pin expansion port used to connect a wide range of external peripherals, such as the MTS310 sensor board used in the adopted experimental setup. For wireless communication IRIS motes are equipped with the AT86RF230 transceiver, a low power ZigBee 2.4 GHz radio supporting IEEE 802.15.4, 6LoWPAN, RF4CE and ISM applications. The platform also includes the DS2401 chip containing a 64 bit ROM (storing a 48-bit serial number, an 8-bit CRC, and an 8-bit Family Code) providing a unique identifier for the device that is used as 16-bit MAC source address.

In order to implement the model described in Section 4.2, the abstract components may be mapped into either hardware units or software.

The *PStg* is implemented in the Flash memory. The storage grows linearly as new symbols are defined on the basis of the previous ones. Special symbols, called *markers*, can be defined to indicate restore points. The execution of a marker causes the *PStg* to rollback to the state it had at the time the marker was defined, deleting all the symbols defined afterwards.

In this model, a variable is merely a symbol that leaves a memory address on the stack when executed. Getting and setting the value is then performed by executing the *fetch* and *store* symbols, respectively. Due to the underlying Harvard

architecture, there is a clear separation between the symbolic identifier, which is part of the word dictionary, and its content which is stored as a volatile value. A constant is instead a symbol that leaves its value on the stack on execution. In this case, both the symbol and its value are kept into the *PStg*. A concrete instance of the *TStg* is thus the RAM memory that holds the content of buffers and variables as well as the stacks.

The Symbol Processor is instead implemented as an interpreter, which is written in assembly, running on the bare MCU. The interpretation of a word thus consists in looking for its name in the *PStg* and executing the code its *xt* points to. Some built-in words make the interpreter perform compiling actions that extend the dictionary by writing to the *PStg*. Loop and selection constructs are provided as built-in words (*compiling words*, cf. Appendix) defined accordingly to the model described in Section 4.2.

AmForth (AmF, 2013), an existing open-source Forth system targeting AVR MCUs, which are at the core of many WSN platforms, is the basis for the implementation of the computational paradigm. Even if it did not support any WSN node architecture, it proved fairly complete.

Although the interpreter design does not exploit any optimization technique, it provides a simple and effective implementation of the abstract architecture, including event handling, which is largely based on interrupts.

The interpreter itself encompasses the function of the *Event-Handler*. Once the interpreter fetches the next instruction, it also verifies the occurrence of events from the hardware devices by testing the T flag of the AVR MCU status register. If the flag is on, it executes the *xt* of the word associated with the interrupt number that is currently active. This way, interrupts from the MCU peripherals are effectively handled by high-level words.

A number of event sources are usually integrated into MCUs, for instance counters and timers, watchdogs, PWM generators, I/O ports and GPIO pins, USARTs, and expansion buses such as I2C, SPI, and USB. Analog interfaces such as ADCs, analog comparators and DACs are also integrated on-chip. Other peripherals, like Ethernet, Wi-Fi, Bluetooth and ZigBee communication modules can be easily added through this wealth of I/O options.

The interaction with the physical world may require reading from the sensory

systems to measure significant parameters. Although low-level components are involved in this task, they are equally mapped into words. For instance, getting back to the example provided in the previous section, the word `temperature` enables the ADC, waits until the reading is available on the ADC data register and then disables the ADC, pushing the raw temperature reading on the stack. The word also sets a flag that identifies the sensor that is currently active. The interpretation of the word `sample` involves checking the value of the flag. Based on this value, the pointer to the sample set of that physical quantity is fetched and the pointer updated. Finally, the word `store` saves the temperature value to the appropriate location.

As hardware peripherals are mapped into symbols and new words can be defined in terms of those previously defined, the development process naturally flows from the definition of low level details towards higher level of abstractions.

In case of very critical time constraints, words can be defined in assembly at runtime by simply using sequences of symbols representing mnemonics and operands.

4.3.2 Communication Modes

By default, a generic node is operated through a wired connection that provides a command-line interface to the interpreter. The interpreter is implemented on the local node as an endless loop that waits for serial input and sends its responses to the serial output. Through the default communication mode a node can be programmed in either interactive or batch mode.

The first mode is useful for rapid prototyping and testing of code. The second mode is useful to install already tested and working application code on the node. In both cases the symbolic code is compiled on board into the execution model code (cf. Section 4.2) by the interpreter.

To provide the user a way to interact with an already deployed network, the platform can redirect the I/O to the radio. While redirection of either input or output is normally performed through specific words, a special frame can be sent to nodes running DC4CD to switch the interpreter input to the radio. This *side channel* permits to awaken deployed nodes and interact wirelessly with the

interpreter running on them.

The second communication mode of DC4CD exploits this feature to inject symbolic code into the interpretation flow of wireless remote nodes through the use of wired bridge nodes. Besides on-line debugging, this operating mode supports nodes retasking, exploratory programming, and fast development of prototypes on the real hardware. As a side-effect, symbolic applications can be built incrementally on remote nodes as if these were physically connected through the serial line.

The third communication mode focuses on the interaction among networked devices, by allowing wireless nodes to inject symbolic code into the interpretation flow of other peer devices. This communication mode is particularly suitable to enable collaborative processing of symbolic knowledge and to support the development of more complex applications. Such a functionality is detailed in Section 4.3.4.

4.3.3 Word Set

The symbolic platform abstracts the application code from the hardware it runs on through the HAL. DC4CD exposes a quite small number of words supporting hardware abstraction, sensing tasks, and networking.

A comprehensive view of all the words defined to build the software environment for application development is provided in Figure 4.2. Words are grouped into word sets according to their specific purpose and interdependencies. Hardware-dependent word sets are depicted as shaded blocks and hardware-independent ones as clear blocks. The Flash memory footprint for each word set is also provided in Figure 4.2.

The interpreter used as the basis of the proposed platform provides just a basic set of symbols to set ports and timers, handle interrupts, and for the management of lower level interfaces. However, it does not offer any networking or other common WSN functionalities, e.g. sensing and actuating tasks.

DC4CD consists of three layers of word sets defined above the interpreter.

The two lower layers hold the words needed for the IRIS mote implementation, except the word set for the construction of valid 802.15.4 MAC frames, while the topmost layer, which is hardware-independent, implements the networking

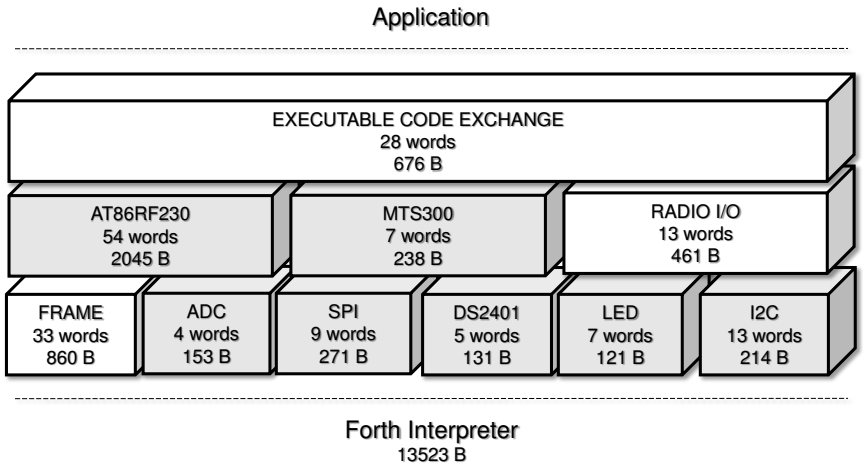


Figure 4.2: DC4CD cross-layer architecture design. The number of words composing each word set and Flash occupation in bytes are also indicated.

abstractions that are described in Section 4.3.4.

This arrangement is not rigid, with fixed boundaries between layers, but it rather results from the adoption of a cross-layer design in which higher-layer words use previously defined lower-layer words. For instance, words of the hardware-dependent *RF230* word set use words of the hardware-abstracted *FRAME* one, for example to set the destination node address before sending a frame.

Although most of these words manage hardware components directly, the expressiveness of their implementation makes the operation of these modules easily understandable and their verification immediate. So coding proceeds through a bottom-up approach in which the most recent words definitions are more abstract than those previously compiled.

The word sets currently composing DC4CD are:

- FRAME: words used to create data frames according to the IEEE 802.15.4 standard. Operations rely on two buffers to store ingoing and outgoing frames;
- ADC: provides the basic functionalities to enable and disable the ADCs, and

to perform the readings;

- SPI: words to enable the communication between the MCU and other SPI devices in both master and slave mode;
- DS2401: includes words to read the DS2401 chip value through the 1-wire protocol;
- LED: word set to deal with IRIS Mote LEDs;
- I2C: primitives to interface the MCU with other devices through a two-wire serial interface, especially used with expansion and prototyping boards;
- AT86RF230: words enabling radio transmission and reception through the on-board RF230 transceiver. This word set depends on the SPI word set as the MCU communicates with the radio transceiver through the SPI interface;
- MTS300: word set driving the MTS300 sensing board. It depends on the ADC word set as ADCs are used for sensor readings;
- RADIO I/O: word set to redirect the standard input and output from the serial line to the radio subsystem. This feature has been exploited to create a remote shell application as detailed in Section 5.3.
- EXECUTABLE CODE EXCHANGE: this word set implements the networking abstractions to support the development of distributed applications. In particular, the high level words enabling the exchange of symbolic code among nodes that will be described in the next section.

4.3.4 Support for Distributed Applications

The design of advanced applications may require the nodes to be context-aware and show adaptivity abilities as well as to cooperate by exchanging more elaborate information than numerical sensory readings, such as qualitative descriptions of a given phenomenon or symbolic rules. However, enabling distributed processing on resource constrained devices requires also architectural efforts.

To incorporate cooperative behaviors through the exchange of symbolic knowledge, DC4CD provides a high-level abstraction mechanism that makes the nodes exchange executable symbolic code.

All DC4CD operating modes are based on such a mechanism for the communication among the bridge and the nodes as well as among peer nodes.

Instead of a thick communication stack, the choice is a networking abstraction that is based on simple constructs. The two high level words `tell:` and `:tell` enclose the executable code to be remotely sent according to the following syntax:

<*dest_addr*> `tell:` <*code*> `:tell`

The destination address of the remote node must be on the top parameter stack before the syntactic construct is evaluated.

The address is not necessarily provided as a literal immediately before the `tell:` word but it can be left on the stack by the previous evaluation of another symbolic expression. The interpretation of the word `tell:` creates a default IEEE 802.15.4 frame and stores the remote node address in the destination address field. It also parses the sequence of symbols that follows and stores it as plain ASCII characters in the outgoing frame until the `:tell` word is encountered. The word `:tell` that closes the syntactic construct for the executable code exchange carries on the frame transmission to the destination node. Communication among nodes is kept at the data link layer.

As a practical example, a command for the bridge node to broadcast the code to acquire a temperature sample could be the following:

```
bcst tell:  temperature sample store :tell
```

The symbol `bcst` leaves on the stack the broadcast address according to the 802.15.4 standard. The words `tell:` and `:tell` are meaningful enough to associate them with the action of telling a node to do something, such as to tell all the nodes to acquire a temperature sample.

This exchange is entirely based on high-level symbols, allowing for frame construction without recurring to any encoding or compression techniques, and without requiring further translation steps as in the case of bytecode injection.

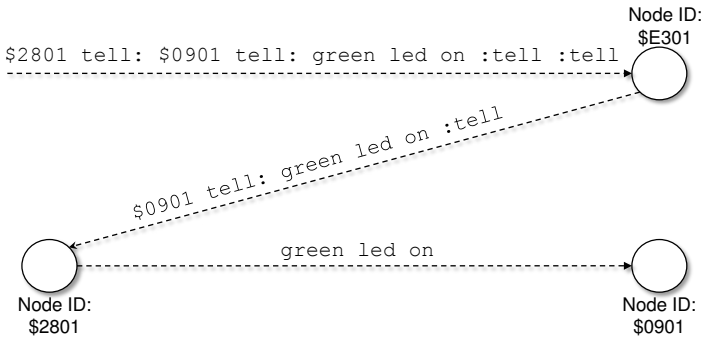


Figure 4.3: Recursive use of the `tell:` <*code*> `:tell` construct for multi-hop symbolic code exchange.

A correct transmission is signaled by an interrupt generated by the radio subsystem.

Special symbols are used as syntactic placeholders to permit the insertion of dynamically computed values into the exchanged code. The insertion at runtime is currently implemented for both 16-bit single-cell and 32-bit double-cell (see Appendix) integer representations, for strings, and for fuzzy truth values using the symbols ∼, ∼∼, ∼`s` and ∼`fm` respectively. Supposing that the topmost item on the local stack is an integer value that must be sent to another node, the sender can be given the command:

<*dest_addr*> `tell:` ∼ `:tell`

Once ∼ is encountered, it is replaced, in the outbound frame, by the number on top of the sender stack. Similar processing applies to the ∼∼, ∼`s` and ∼`fm` symbols.

Nested usage of this syntactic construct allows to convey symbolic code from a source to a destination node, by making intermediate nodes acting as forwarder to other nodes in the route to destination. Figure 4.3 clarifies this approach whose semantics can be summarized as *to tell a node to tell another node to do something.* The code above the arrows is the payload interpreted by the destination nodes.

According to Forth conventions, square brackets enclosing symbols indicate an alternative to either the compile-time or execution-time behavior of the symbol without brackets. In this case, when code exchange is required inside a word definition, the `[tell:]` <*code*> `[:tell]` construct must be used instead. This

ensures that what is enclosed between `[tell:]` <*code*> `[:tell]` is stored inside the new word definition. However, at runtime, once the new word is interpreted, `[tell:]` <*code*> `[:tell]` performs exactly as `tell:` <*code*> `:tell`.

The executable code exchange paradigm also allows to directly exchange assembly code enclosed within the built-in words `code` and `;code`.

4.4 Case Study

In the following, two applications for DC4CD are presented. The first aims to show how it is possible to make an already deployed network to accomplish new tasks without programming the nodes in advance. The second concerns the implementation of symbolic reasoning abilities on board of resource limited devices in the development of a distributed event detector based on fuzzy inference.

4.4.1 In-network Distributed Data Aggregation

Data aggregation is usually performed in WSNs through centralized schemes based on established protocols, such as CTP, and rigid rules for both communication and processing.

This section shows how the high-level constructs of DC4CD allow for the implementation of generic in-network distributed data aggregation tasks that are both unknown to nodes at the deployment time and changeable at runtime.

The key of the proposed approach is the exchange of symbolic programs through the construct introduced in section 4.3. The task is started on deployed nodes having no other abilities than those provided by DC4CD.

An initial setup phase is then started to establish a simple order among nodes according to which the executable code crosses the network. In the resulting linear topology, two-way links connect each node to its successor.

Then, pure symbolic code injection makes the nodes cooperatively perform the aggregation of physical quantities –e.g. by computing the minimum, the maximum, the average and so on. Each of these tasks can be injected into the network at any time without programming the network before deployment.

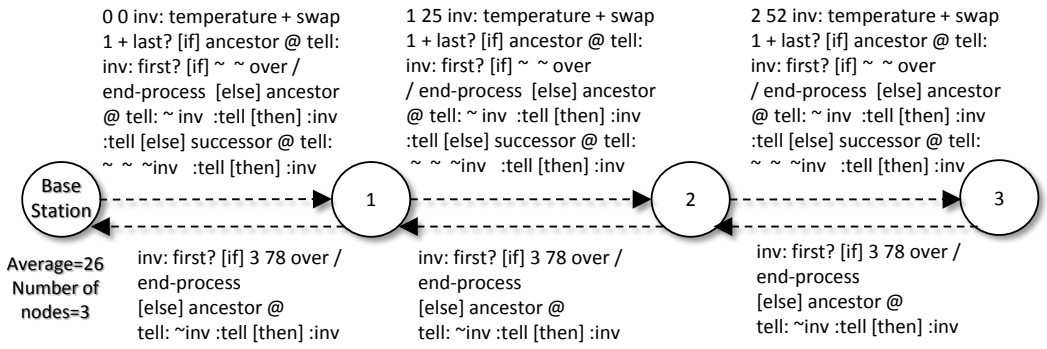


Figure 4.4: Symbolic code flowing in the network as plain text for the distributed temperature average computation.

For the sake of clarity, the case of asking the network about the average temperature of the environment is considered.

Let us assume that the nodes are in the same radio range and share the same notion of time. The example intentionally skips networking details and efficiency aspects to show how collaborative behaviors can be implemented by exploiting the on-board symbolic evaluation in a fully distributed way.

The other available interpreter-based architectures run programs stored into RAM (see Section 4.1). Besides permitting to run stored programs, the proposed approach also allows to inject pure symbolic code into the network. This way, the network can be made perform further tasks besides those already required through static coding.

For instance, it is not required to program in advance each node to compute cooperatively the minimum, the maximum or the average value of a certain physical quantity.

As an example, the plain text program to compute the average temperature, which is exchanged by nodes according to a linear topology, is shown in Figure 4.4.

At each step, one device measures the temperature, aggregates the values – i.e. adds the measured temperature to the last received aggregate– increments the number of nodes, and checks whether it is the last of the chain. If it is not, it propagates the updated number of nodes, the current aggregate and the same program to its successor. Otherwise, the symbolic code to retrieve the temperature

aggregate and the number of nodes is sent to its ancestor.

The symbolic program injected in the network is the following:

```
0 0                  \ Initial values for number of nodes and temperature
inv:                 \ the invariant part of the message starts here
temperature +        \ measure temperature and add it to TOS
swap 1 +             \ increment the number of nodes by one
last? [if]           \ if you are the last of the chain
  ancestor @ tell:   \ send your ancestor the invariant code
  inv:               \ from here to the following :inv that commands:
  first? [if]                   \ ``If you are the first of the chain
    ~ ~                         \ you will find the number of nodes and
                                \ the current aggregate on your stack,
    over / end-process          \ thus compute the average and terminate;
  [else]                        \ else, as you are not the first of the chain
    ancestor @ tell: ~inv :tell \ send your ancestor the inner invariant code
                                \ to do the same as you did,
  [then]
  :inv :tell                    \ which ends here.''
[else]               \ else, as you are not the last of the chain
  successor @ tell:  \ send your successor
  ~ ~ ~inv :tell     \ your aggregate, number of nodes and
[then]               \ the same outer invariant code you are executing now,
:inv                 \ which ends here
```

The syntactic construct `inv:` ⟨*code*⟩ `:inv` is used to mark an invariant part of the incoming message. The propagation of invariant code uses the marker ∼`inv` within `tell:` ⟨*code*⟩ `:tell` for its runtime substitution. The symbols `[if]`, `[then]` and `[else]` are the execution-time counterparts of `if`, `then` and `else`, which can be only used in word definitions.

In order to ask the minimum temperature value sensed by the network, the code to be injected is slightly different, as it includes some built-in words for stack manipulation, i.e. `nip, swap, dup`. The syntactic construct to manage invariant code is used in this case too:

```
0 800                \ Initial values for node ID and minimum temperature
inv:                 \ the invariant part of the message starts here
temperature          \ measure temperature
2dup < [if]          \ if temperature is higher than current minimum
  drop               \ discard the temperature value
[else]               \ else, if temperature is equal or lower
                                 than current minimum
  nip nip id @ dup   \ perform stack operations to put node ID
                                          and minimum on top
  last? [if]         \ check if you are the last of the chain
```

```
    ancestor @ tell:    \ if so, send you ancestor the invariant code
    inv:                \ from here to :inv that commands:
    first? [if]                     \ ``If you are the first node
      ~ ~                           \ you will find the ID and the minimum
                                    \ temperature on your stack
    [else]                          \ else, if you are not the first of the
                                        chain
      ancestor @ tell: ~inv :tell   \ send your ancestor the same
                                                inner invariant code,
    [then]                          \
   :inv :tell                       \ which ends here.''
  [else]                \ else, if you are not the last node of
                                            the chain
    successor @ tell:   \ tell your successor
    ~ ~ ~inv :tell      \ node ID, current minimum value and the same
                        \ outer invariant code,
  [then]
  [then] :inv           \ which ends here.
```

As it is presented, the application runs without requiring any code to be stored on nodes. Adopting a different, but equally plausible, design path, the code could be refactored in more words. These, in turn, would then be sent to nodes before processing starts, and locally compiled at the sole expense of a few bytes of Flash memory. For instance, a more compact and abstract version of the application code could be:

```
0 0 inv:
temperature + swap 1 +
last? [if] retrieve
      [else] propagate [then]
    :inv
```

The words `retrieve` and `propagate` could have been pre-defined as such:

```
: retrieve
  ancestor @
    [tell:]
        inv: first? if
               ~ ~ over / end-process
               else
               ancestor @ tell: ~inv :tell
               then
        :inv
    [:tell] ;

: propagate
  successor @ [tell: ] ~ ~ ~inv [:tell] ;
```

4.4.2 Distributed Fire Detection

While the previous section presented a practical example of collaborative data aggregation without any stored code, the goal of this section is to discuss the implementation of a more complex application needing the inclusion of further abilities, such as inference, on resource-constrained embedded systems. To this purpose, the code for DC4CD of a distributed fuzzy system for fire detection (Marin-Perianu and Havinga, 2007) shows how cooperative mechanisms improve node decisions through the fusion of local sensory readings with those of the neighborhood.

The fuzzy inference system for DC4CD required the definition of 31 words with a total memory usage of only 6 bytes of RAM and 863 bytes of Flash storage.

In this fire detection application each node regularly measures temperature and smoke, applies fuzzification to raw values as well as to their variations, and asks *opinions* to the neighborhood. A node broadcasts the code to make the others waiting for a random time before dispensing its *opinion*, as follows:

```
bcst tell: 0 1000 random ms   opinion dispense :tell
```

Remote nodes wait for a random time of up to one second then execute `opinion`, which leaves on the stack the addresses of the four membership functions bound to temperature and smoke. Finally, the word `dispense` creates a reply packet containing fuzzy truth values as well as the code to treat them on the requiring node. The word `dispense` is defined as follows:

```
: dispense
   reply [tell:] 1 nodes +!   ~fm fcount +!
   ~fm fcount +!   ~fm fcount +!   ~fm fcount +! [:tell] ;
```

The word `~fm` is a placeholder for the runtime inclusion of the pair <*truth_value, membership_function*> in the outbound frame. The word `fcount` leaves on the stack the address of the counter storing the partial aggregated sum of fuzzy truth values related to that membership function, which is used later to compute the cardinality of the fuzzy set. The word `+!`, which increments by the second topmost item on the stack the value the address on top of the stack points to, is used to update the number of `nodes`, as well as the partial counters by fuzzy truth values received by neighbors. The code thus increments on the requesting node the num-

ber of nodes, as well as the partial aggregate, i.e. a *fuzzy counter*, of fuzzy truth values sent by another node. After the timer for neighborhood responses expires, the node uses the *fuzzy counters* to compute the fuzzy set cardinality, and to apply the `most` fuzzy operator on it, as described in (Marin-Perianu and Havinga, 2007). Then it executes fuzzy rules that incorporate neighborhood opinions. As an example of the implementation, one of the fuzzy rules taken by (Marin-Perianu and Havinga, 2007) in DC4CD looks like:

```
temp.low @ smoke.low @ &
temp-neighbor.low @ smoke-neighbor.low @ & &
dtemp.low @ dsmoke.low @ & & => fire.low
```

Besides the use of the `@` word to *fetch* the value of the fuzzy membership function and the fuzzy *and* operator (`&`) in postfix notation, the rule is pretty self-explanatory.

Finally, fuzzy variables as well as fuzzy membership functions can be sent or modified by the bridge node at runtime without any recompilation on remote nodes.

4.5 Experimental Evaluation

All the proposed interpretation-based platforms in literature run on some general-purpose OS (see section 4.1). As DC4CD integrates the role of both an OS and a symbolic expression evaluator, in order to provide a detailed and fair experimental evaluation, the latter is split it in two parts.

In the first, DC4CD is compared to the most prominent OSs for WSNs, TinyOS and Contiki (Farooq and Kunz, 2011), assessing the memory requirements of some test applications, as well as the DC4CD interpretation overhead with respect to native implementations. Actually, this comparison is disadvantageous to DC4CD as the symbolic interpreter cannot be detached. Results thus provide an estimate of the baseline resource requirements for all the platforms, indeed a worst-case one for DC4CD.

Then, DC4CD is compared to other reference interpretation-based architectures aimed at the development of symbolic applications such as Maté (Levis and

Culler, 2002), T-RES (Alessandrelli et al., 2013) and TakaTuka (Aslam et al., 2010). For these tests the same benchmark applications reported by the authors of the respective platforms have been used.

4.5.1 Comparison with General-Purpose Operating Systems

The Flash usage of DC4CD, TinyOS-2.1.2 and Contiki have been compared via the microbenchmark test suite used in (Dong et al., 2011), targeting IRIS motes. In order to provide a more precise estimate of the resource usage in real-world applications, the widely adopted CTP (Bucur et al., 2014; Zuo et al., 2015) has been used as a further benchmark. Even if CTP is a centralized protocol, it can be useful even in distributed scenarios, e.g. to integrate a WSN with the IoT through a gateway node. Memory usage due to the platform is the *Baseline*, while *Increment* refers to Flash memory occupation of the application. Results are reported in Figure 4.5. Benchmark applications are indicated in increasing complexity order. Although Contiki includes by default a complete TCP/IP stack, in order to have a fair comparison, an image compiled with the only low-level 802.15.4-frame-based Rime as a communication stack has been used in these tests. In addition, the broad adoption of Rime in WSN applications gives the comparison more realistic conditions. (Evangelatos et al., 2012; Fortuna and Mohorcic, 2014; Chatzigiannakis et al., 2016).

DC4CD resulted more compact than Contiki with regard to Flash usage, as Contiki memory footprint is almost constant for all the applications under scrutiny. However, DC4CD showed a larger memory overhead than TinyOS in the first four benchmarks. Such a result is primarily due to the virtual machine above which DC4CD lies, which measured slightly less than 10 KB, about the 9% of the entire Flash, while the memory occupation increment due to DC4CD word sets and to the application code was just a few bytes. Conversely, DC4CD Flash memory usage was lighter than TinyOS in the *RadioCountToLeds* test, as shown in Figure 4.5. Still, TinyOS and Contiki benchmark applications are statically programmed as any modification in the application code implies recompiling and reflashing the entire binary image.

Figure 4.6 provides an assessment of RAM usage for the same suite of test applications. As in the case of the Flash occupation, the virtual machine RAM footprint (287 bytes) constitutes a baseline for DC4CD RAM requirements. The *Oscilloscope* test requires additional 28 bytes of RAM to store the acquired temperature sensory readings before sending them to the base station. For the *CTP* test, a symbolic implementation of the protocol is compared, in order to show how non-trivial processing and communication algorithms based on high-level messages may be designed using the proposed methodology. This actually has also conceptual implications, as the symbolic CTP protocol is able to *collect* both data and code from nodes, as shown in Figure 4.8. From the practical point of view, packets do not need to follow the compact but rigid binary structure used in reference implementations that only need to send a few sensor readings at once. On frame reception what is enclosed between `c<` and `>c` is enqueued by executing the word `enq` and then an ack `reply` message is sent to the sender. The first five numbers are respectively the *congestion status*, *thl*, *etx*, *origin* and *sequence number* parameters required for the CTP data packet header. Once the data packet is ready to be forwarded, its content is interpreted, except for the application payload, which is the symbolic code between `p:` and `:p`. The word `hdr` updates the parameter values prior to forwarding, while the word `rt-lp?` detects routing loops as described in (Gnawali et al., 2013). In these tests, a queue of 13 packets is considered, as in the reference TinyOS implementation. However, as the proposed implementation permits to send symbolic and dynamically generated payloads having variable length, packets in the queue are allotted 128 bytes each, the maximum amount of data that can be included in a single frame. The relatively large increment of RAM usage of the proposed implementation can almost totally be ascribed to the queue. For the other benchmark applications no additional RAM memory to that used by the virtual machine was required. Contiki RAM usage measurements in Figure 4.6 only takes into account global data and does not include stack and heap runtime consumption. In spite of this, also in this comparison DC4CD applications required less memory than their Contiki counterparts.

The number of code lines required by each benchmark are also. As reported in Figure 4.7, the DC4CD implementations showed higher code compactness than the respective versions for TinyOS and Contiki.

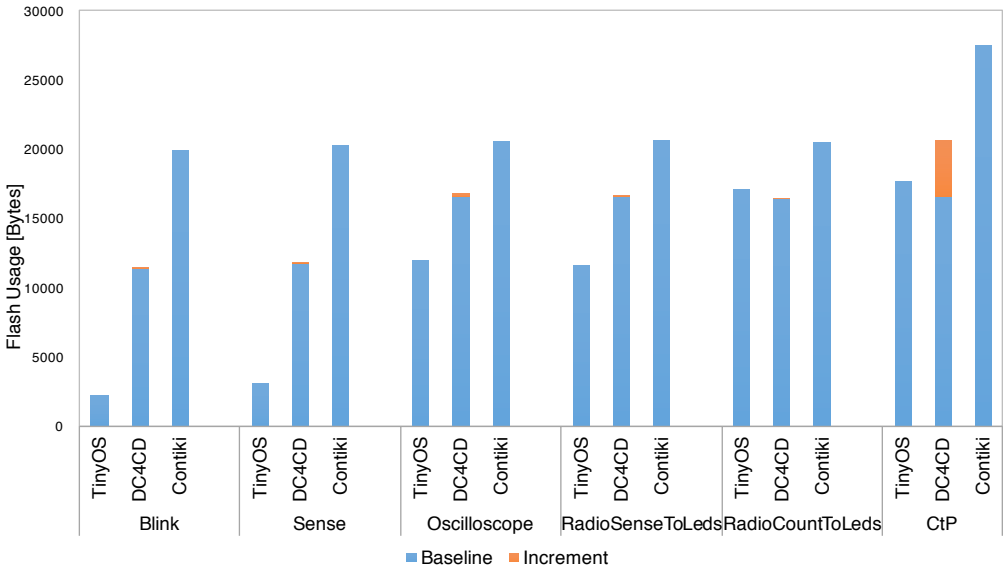


Figure 4.5: Flash memory footprint for the suite of benchmark application in TinyOS and Contiki vs. DC4CD

After evaluating memory consumption for IRIS motes, the interpretation overhead has been assessed. To this purpose, the test considers the representative word `fill` that initializes an array of 2048 items with a given value, and the turnaround time over 10,000 executions has been measured.

The test confirms that the interpretation process remarkably slows down the execution process. The average turnaround time in DC4CD was 33 times slower than the Contiki native implementation. This result is due to the indirect threaded compiler design. However, this value provides an estimate of the time required for the interpreter to solve indirection and to get the address of the machine code to be executed.

In order to support time-critical code, the platform also includes high-level constructs for the injection of assembly code into a remote node. To speed up the execution time of DC4CD, the remote node is provided with an assembly lan-

Table 4.1: Interpretation overhead

	Initialization of a 2048 items array		
	Interpreted Version	**Assembly Version**	**Contiki**
Turnaround time [s]	0.124	0.005088	0.002617

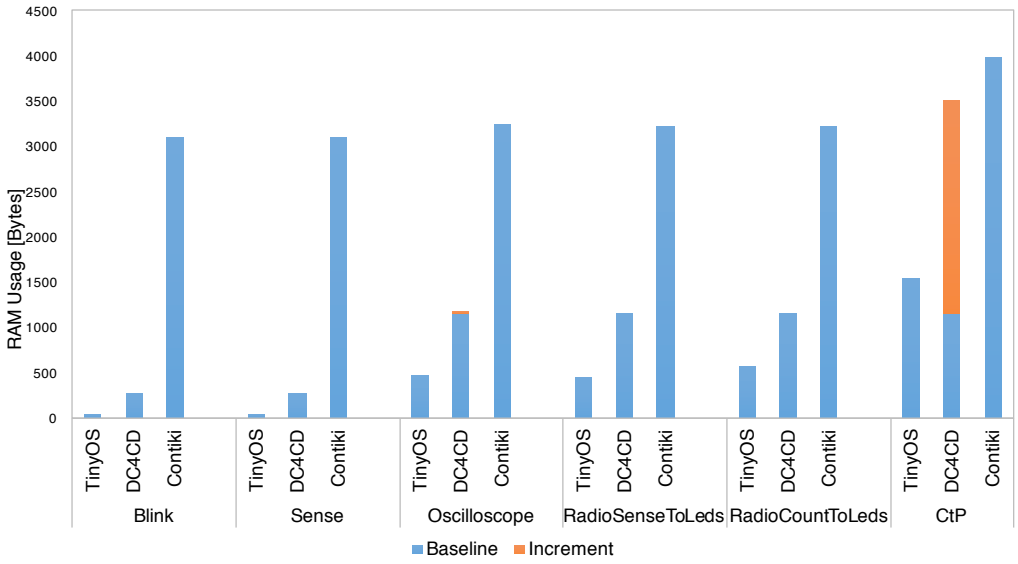


Figure 4.6: RAM memory footprint for the suite of benchmark application in TinyOS and Contiki vs. DC4CD

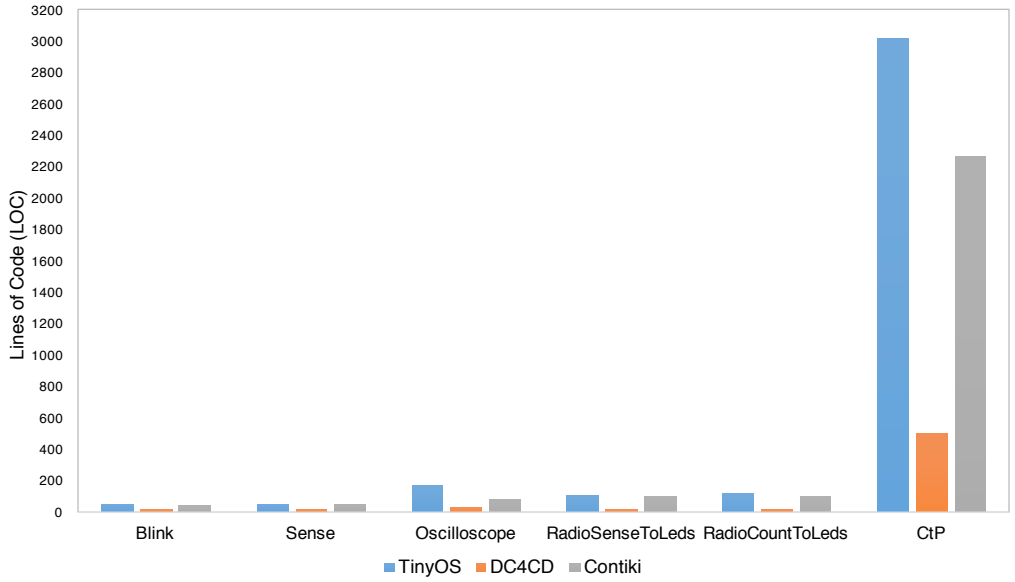


Figure 4.7: Lines of code for the suite of benchmark applications in TinyOS vs. DC4CD

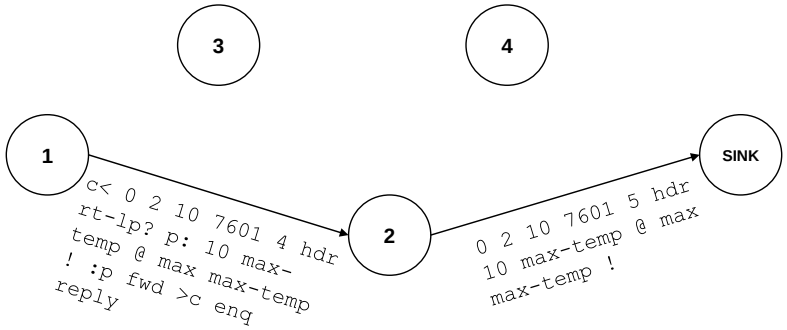


Figure 4.8: Symbolic CTP implementation. Frames to sink contain CTP data, CTP code, and application code. CTP header data is coded in the first five numeric values. The words `hdr`, `rt-lp?`, `fwd`, `enq`, `reply` belong to CTP code. The application code, between `p:` and `:p` includes a sensory reading and the code to make the sink update the maximum collected temperature value.

guage definition for the word `fill`, and again the time for completion over 10,000 executions has been measured. Results are provided in Table 4.1. As expected, by reducing the interpretation overhead the execution time was comparable with the Contiki implementation. The difference between the values accounts for the interpretation cost of the instructions for the test loop.

To conclude comparisons with general-purpose operating systems, results concerning DC4CD power consumption have been also reported. A similar approach to the PowerTrace energy profiler of the Contiki development environment (Dunkels et al., 2011) has been used. It consists of a linear model taking into account the time spent by the various system components – e.g. radio, CPU, and sensors – while running the application. Contrarily to the PowerTrace profiling, which works in a simulated environment, in this case, measurements were performed right on the motes, though.

While the application was running, a timestamp was recorded for each switch-on and switch-off of each component. The timestamps were then used to evaluate the total time of operation for each component. The DC4CD profiling system could rely on a 200 μs resolution timer for this evaluation.

To take into account the energy required to interpret the high level code, the application ran in a busy loop.

Table 4.2: Percentage of CPU activity time for application code

Benchmark	DC4CD	Contiki
Blink	0.072%	0.003%
Sense	0.208%	0.090%
Oscilloscope	0.433%	0.023%
RadioSenseToLeds	2.512%	3.654%
RadioCountToLeds	2.408%	3.486%

The time spent by the CPU executing application code with respect to the total time of execution is reported in Table 4.2 for all the benchmarks.

4.5.2 Comparison with other Interpreted Architectures

DC4CD was compared to representative interpreted architectures such as Maté, T-RES and TakaTuka. The platform has been evaluated through the same test-benches reported by the authors of the platforms mentioned above.

Some specific efforts were needed to compare DC4CD with Maté. This platform only runs on TinyOS 1.x but, unfortunately, the IRIS hardware is not supported by this version of the host OS. Maté has been compiled for MICAz, which is the closest to IRIS among the hardware supported by the platform, with the main differences between the two kinds of hardware consisting in the on-board radio subsystem, and FLASH and RAM memory amounts on the MCU. Actually, some more work had to be done to align versions of GCC and the nesC compilers, and of the AVR libraries in order to obtain a usable virtual machine. This is indicative of how involved the development process even for simple devices may become by adopting classic toolchain-based approaches.

In order to provide a rough estimate of the computational performance of the platform, the same test as in (Levis and Culler, 2002), which runs a tight loop simply incrementing a counter in a five-second time interval, has been carried out. To assess the number of instructions per second (IPS), the number of iterations in the time interval is used as a metric. The comparison was thus between Maté VM instructions and DC4CD words. Results show that the average IPS for DC4CD is 3.4 times that of Maté.

Results presented in Table 4.3 show that DC4CD had lower memory foot-

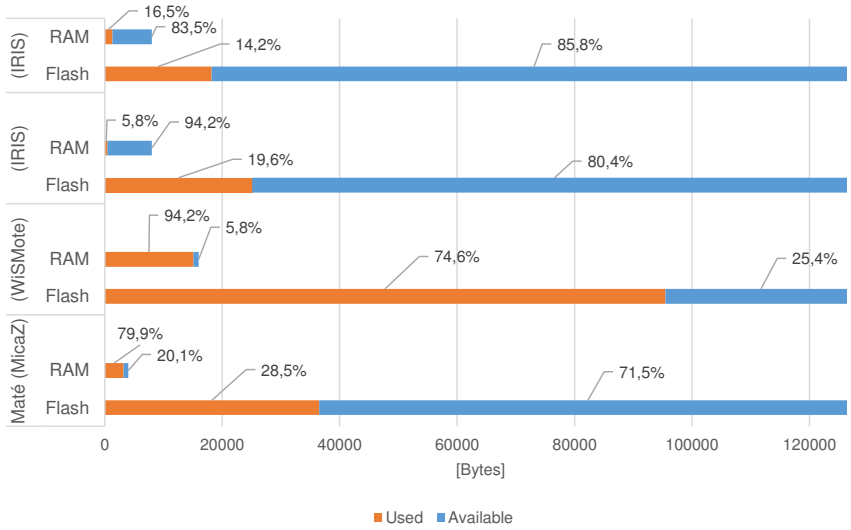


Figure 4.9: Relative memory usage comparison of DC4CD along with representative interpreter-based architectures.

print than other interpretation-based architectures, even when including all the word sets indicated in Figure 4.2. Figure 4.9 confirms that the implementation of the interpreter above a general purpose operating system occupies much of the available memory, as in the case of Maté and T-RES. Although the hardware is different, the chart displays the relative memory usage with respect to the available amount. As scripts are stored in RAM, not enough space is left to the development of complex applications, even with the double-sized RAM of WiSMotes.

Furthermore, to compare the proposed architecture with a representative implementation of the Java Virtual Machine on tiny devices,TakaTuka has been compiled on top of TinyOS-2.1.2. Results in Table 4.3 regarding TakaTuka refer to the Flash and RAM usage of a minimal Java program consisting of an empty main method, which can be considered as the baseline for memory requirements of applications for this platform. Despite TakaTuka adopts optimizing techniques to reduce memory requirements for the Java classfiles and for the JVM interpreter, DC4CD proved more efficient, as shown in Table 4.4, in the three benchmark applications reported by the authors. For a more precise comparison, DC4CD results are presented as a sum of the platform and the application memory usage. DC4CD RAM usage exceeded TakaTuka requirements in the *radioApp* application. How-

Table 4.3: Memory requirements of interpretation-based platforms

	Maté - Bombilla	**T-RES**	**TakaTuka**	**DC4CD**
Flash [Bytes]	36506	95455	25122	18714
RAM [Bytes]	3196	15078	466	1321
Target platform	MICAz	WisMote	IRIS	IRIS

Table 4.4: DC4CD vs. TakaTuka

	Flash Usage [bytes]		**RAM Usage [bytes]**		**Lines of Code**	
	DC4CD (platform+app)	TakaTuka	DC4CD (platform+app)	TakaTuka	DC4CD	TakaTuka
Blink	11399+87	26440	289+0	470	55	59
radioApp	16370+112	37848	1159+0	846	112	116
binarySearch	11399+250	30608	289+10	484	56	63

ever, RAM usage in DC4CD was not so critical as in TakaTuka, which uses RAM to store injected applications.

Finally, to assess the cost of retasking already deployed nodes, the platform has been compared to T-RES. The network traffic generated by the code installation process provided an estimate of the cost to inject new code on remote nodes. This is considered by the authors of Maté as a metric for the energy consumption evaluation, since the compactness of the code to be injected directly affects the time spent in transmission.

For this evaluation, the injection of a monitoring application is considered. It consists of two nodes that sense the temperature at regular intervals, while another computes the average temperature. The value is then sent to the actuator node. Whenever the average temperature exceeds a fixed threshold, the actuator node switches the heater off, otherwise the heater stays on. Three different tasks –sensing, averaging and actuating– must be injected at runtime on entities of three different kinds. For the sake of homogeneity, as T-RES targets WisMote nodes, the installation cost has been evaluated in a platform-independent way, by considering the number of bytes that cross the network for the retasking purpose. T-RES has been evaluated using the Contiki network simulator Cooja, while the reprogramming cost of an already deployed DC4CD node was tested in a real testbed deployment and the number of data bytes composing a frame was computed.

Figure 4.10 reports the code of the averaging task for T-RES and DC4CD, respectively. The word `avg.isr` is executed periodically by the node. This word

```
from tres_pymite import *

class state:
  def __init__(self):
    self.t1 = -273.15
    self.t2 = -273.15

print "Simple Average:",
s = getState(state)
tag = getInputTag()
if tag == "Temp1":
  s.t1 = getInput()
else:
  s.t2 = getInput()
saveState(s)
if (s.t1 > -273.15) and (s.t2 >
    -273.15) :
  setOutput ((s.t1 + s.t2) / 2)
```

```
variable temp1
variable temp2
variable actuator

: -273<>?
  -273 > swap -273 > and true = ;

: temperature-avg
  2dup -273<>? if + 2 /
  else 2drop then ;

: average
  temp1 @ temp2 @
  temperature-avg ;

: act
  actuator @
  [tell:] ~ switch-heater
  [:tell] ;

: avg.isr
  temperature temp2 !
  average act ;

: start-avg
  ['] avg.isr timeout
  timer3.init timer3.start ;
```

Figure 4.10: Averaging Task in T-RES vs. DC4CD

Table 4.5: Task installation cost. DC4CD vs T-RES.

Task	**T-RES [bytes]**	**DC4CD [bytes]**
Sensing	-	700
Averaging	2903	186
Actuating	2260	123

samples the temperature, averages the sample with the temperature previously received by another node, and sends the actuator the code to switch the heater on or off according to the computed average temperature.

Results in Table 4.5 show the cost for the injection of the application in both T-RES and DC4CD. Such results are in part due to the CoAP and IPv6 protocols T-RES relies on. However, the implementation of the Python virtual machine and the communication protocols almost saturates the RAM memory, and prevents the injection of scripts that are more complex than those discussed here. On the other hand, Table 4.5 remarks the compactness of the symbolic code for DC4CD, which fits in a few 802.15.4 frames. T-RES applications are programmed by customizing predefined objects so to specify a processing function. These objects already include sensing so there are no separate figures for the sensing task in T-RES in Table 4.5. However, T-RES requires an initial effort to configure a T-RES node as data input source to another node performing a certain task –e.g in the averaging task the temperature measurement from another node is required. To install a new task, the application script must be cross-compiled into Python bytecode that is then sent to the target node. For instance, the application code specifies the averaging task but does not include the dataflow execution scheme that is enabled by defining node connections via a CoAP plugin in the user browser.

In DC4CD, the injected code is stored in the Flash memory. Collaborative behaviors, the interaction among nodes as well as any other collaboration model are included in the code itself so that no additional packets need to be exchanged.

4.6 Discussion

The examination of the existing literature showed that interpreted architectures proposed so far, although allowing reprogramming remote devices, are rather

resource-hungry and do not provide natively any support for distributed computing CPS applications.

DC4CD supports traditional CPS functionalities, e.g. sensing and actuation tasks, while allowing to extend conventional node capabilities by exploiting the symbolic computation paradigm. The proposed platform runs programs as sequences of high level words that abstract even low level hardware details, as well as interrupt handling, thus ensuring interoperability. Besides remote node retasking, DC4CD supports the injection of symbolic code among resource constrained devices to support the accomplishment of high level goals according to any cooperative scheme.

The networking abstraction, which consists on simple words especially conceived for the exchange of executable code among nodes, has been described as well as how pure assembly code can be sent to avoid interpretation overhead and for kernel modifications. From the above considerations, DC4CD offers an approach differing from those in which programming precedes deployment and nodes are able to perform just the tasks for which they were designed. In fact, the ability to run programs that are included in radio frames makes the programming phase quite dynamic. It is worth dwelling on the security issue that is slightly exacerbated by the interpretation of what arrives as a radio input, as a malicious node could send bad code that would be executed and the entire distributed computation could be compromised. To prevent the injection of malicious code among nodes, further words – implementing deferred execution – have been included to decrypt radio inputs before interpretation occurs and to encrypt symbolic code before sending it to remote nodes. However, the specific protocol is the responsibility of the programmer.

Experimental evaluations proved that the proposed software architecture is highly efficient in terms of memory footprint and code compactness, thus allowing for the development of sophisticated applications without incurring resource overflow issues.

In conclusion, DC4CD has been proposed as an alternative platform that is fully controlled by the designer. The application designer can build dynamic applications, including complex protocols atop it, and intelligent behaviors can be implemented in an efficient way even on-board resource constrained devices.

Chapter 5

Applications

According to the IoT vision (Atzori et al., 2010), all kinds of devices, although computationally limited, might be used to interact with people or to manage information concerning the individuals themselves (Guo et al., 2013). Besides reactive responses on input changes, the whole network may exhibit more advanced behaviors resulting from reasoning processes carried out on the individual nodes or emerging from local interactions. However, nodes' constraints leave to system designers many challenges to face, especially when distributed applications are considered (Martorella et al., 2014). Conventional programming methodologies often prove inappropriate on resource constrained CPS devices, especially when knowledge must be treated with a high level representation or changes of the application goals may be required after the network has been deployed (Kortuem et al., 2010). Moreover, the implementation of intelligent mechanisms, as well as symbolic reasoning, through rigid layered architectures, reveals impracticable on CPS resource constrained devices such as those commonly used in WSNs. Often this issue addressed by adopting an intelligent centralized system that uses WSNs as static sensory tools (De Paola et al., 2014b).

This chapter describes the feasibility of the proposed approach as well as its applicability in developing real applications. Section 5.1 describes an application to make resource constrained nodes locally reason about their position with respect to thermal zones of the deployment area by extending the symbolic approach characterizing the programming environment, which has been presented in chapter 4,

with Fuzzy Logic. Section 5.2 describes the implementation of a distributed Ambient Intelligence (AmI) application to control an HVAC system according to the user thermal comfort. The Fuzzy Logic formalism has been extended to support distributed update of values and evaluation of rules. Finally, the development of a remote shell application for interacting with and programming remote nodes is presented in Section 5.3.

5.1 Inferring the Node Distribution according to Thermal Zones

In this section, the proposed system is used to provide CPS resource constrained devices with reasoning abilities by implementing a Fuzzy Logic symbolic extension on deployed nodes at runtime. In particular, supposed the environment be divided into thermal zones, deployed nodes are able to classify themselves into clusters as well as to discover the neighborhood distribution with respect to thermal zones. As detailed in Appendix, the proposed environment also provides a Command Line Interface (CLI) allowing for interactive development on already deployed nodes. The experimental setup thus consists of sensor nodes pervasively deployed in the physical environment and reachable only wirelessly, and a bridge node connected to the user computer. The designer interacts with this node through the CLI and can send messages to the other nodes in the network.

5.1.1 Local Symbolic Reasoning with Fuzzy Logic

In the proposed programming environment, purely reactive behaviors can be easily implemented on the remote nodes by sending them the sequence of words to be executed if certain conditions are met. Considering the following command given through the Command Line Interface (CLI) of the bridge node:

```
bcst tell: close-to-window? [if]
red led on [then] :tell
```

This command broadcasts –the word `bcst` leaves the reserved address for the purpose on the stack– the code between the `tell:` and `:tell` words. Once received,

each node executes the word `close-to-window?` to evaluate if it is close to the window and, if so turns the red LED on. The word `close-to-window?`, already in the dictionary, performs temperature and luminosity measurements and checks if both sensory readings are above a predefined threshold. As it can be noticed, the code is quite understandable, although all the words operate just above the hardware level by setting ports or enabling the ADCs to read temperature and light exposure. For the sake of showing how it is possible to incorporate new abstractions to support intelligent applications here a Fuzzy Logic extension is introduced. Fuzzy Logic has the peculiarity to be appropriate to implement approximate reasoning in several contexts as well as for machine learning purposes (Navara and Peri, 2004). A classic Forth Fuzzy Logic implementation (VanNorman, 1997) has been adopted and then modified to make it run on the Harvard architecture AVR microcontroller used in the IRIS platform. Finally, the original implementation has been enriched with the possibility to exchange fuzzy definitions and evaluation among nodes.

The wordset to enable high-level fuzzy reasoning on IoT resource constrained devices is provided in Table 5.1 and allows for the creation of fuzzy input/output variables, for the definition of the related membership functions, for fuzzification, for rule evaluation and for defuzzification processes. Differently from (VanNorman, 1997), to create a new fuzzy variable the word `fvar` has been included to be used according to the following syntax:

<*min_val*> <*max_val*> `fvar` <*name*>

where <*min_val*> and <*max_val*> represent the definition domain of the fuzzy variable and <*name*> is the name associated with the new variable. Differently from <*min_val*> and <*max_val*> values that are expected to be on the stack, the variable name is provided at runtime. When this construct is executed by the node interpreter, a new entry named <*name*> is created in the dictionary, which is located in Flash memory, while a five cells structure is allocated in RAM. As illustrated in Figure 5.1, a fuzzy variable can be thought of as a sequence of fields. The Forth code to create this structure is self-explanatory:

```
begin-structure fv
   field: fv.crisp
```

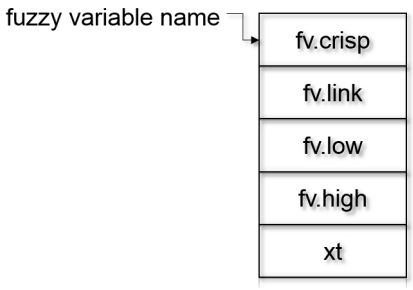


Figure 5.1: The definition of a fuzzy variable include a new entry in the Flash word dictionary and allocates five contiguous cells in RAM memory as a sequence of fields. The first cell stores the crisp input value, while the link field contains the address of the first defined membership function related to the fuzzy variable. The following two cells store the validity range, while the last one stores the execution token (*xt*), i.e. the address of the word to sense the physical quantity associated with the fuzzy variable. Once the fuzzy variable name is used, the address of the first field is fetched on top of stack.

```
   field: fv.link
   field: fv.low
   field: fv.high
   field: xt
end-structure
```

Once the word `fvar` is executed, a generic `fv` structure is instantiated and <*min_val*> and <*max_val*> values are stored in the `fv.low` and `fv.high` fields. The first field stores the crisp input value, and it is followed by a link field, i.e. the membership function list associated with that fuzzy variable. The following two fields contain the validity range, i.e. the minimum value and the maximum value allowed for the crisp input. Finally, as the main goal is to allow the nodes to reason about sensory data, the last field contains the address of the word to perform the measurement of the physical quantity associated with that variable.

Let us define two fuzzy variables, `temp` and `lightexp`. The last field of `temp` stores the address of the word `temperature`, while the address of the word `luminosity` is the last field of `lightexp`. The words `luminosity` and `temperature` have been already introduced in the previous section.

Similarly, to create a membership function, the word `member` expects on the stack four control points which determine the shape of the membership function and its name is provided at runtime according to the following syntax:

<bottom-left> <top-left> <top-right> <bottom-right> `member` *<name>*

As a fuzzy variable, also a membership function is a generic structure composed of several fields:

```
begin-structure membership
   field: fval
   field: link
   field: lm
   field: lt
   field: rt
   field: rm
   field: ls
   field: rs
  end-structure
```

The first field contains the truth value resulting after the fuzzification process, while the second field stores the address of the next membership function. Essentially, a fuzzy variable and its membership functions are implemented as linked list. Membership functions are trapezoidal and therefore four control points are stored in the appropriate four successive fields, left-most (lm), left-top (lt), right-top (rt), and right-most (rm). Finally, two further memory cells are required to store the left slope (ls) and the right slope (rs) of both sides. When the first membership function is defined, the fuzzy variable link field stores the address of the newly created membership function. As the word `member` is executed, the four control points on top of stack are stored in the appropriate fields of the `membership` structure along with the left and right slopes. Figure 5.3 shows the code to define the fuzzy variable related to light exposure named `lightexp` and the related membership functions according to the words described previously.

Moving on with the initial example in which a node evaluates its proximity to a window, in place of two crisp variables, the fuzzy variables `temp` and `lightexp`

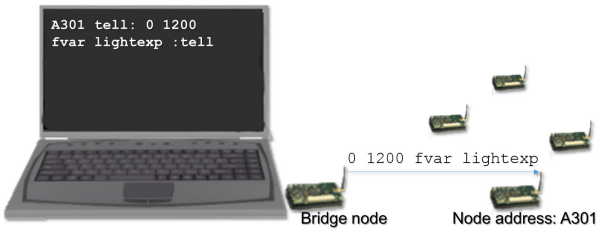
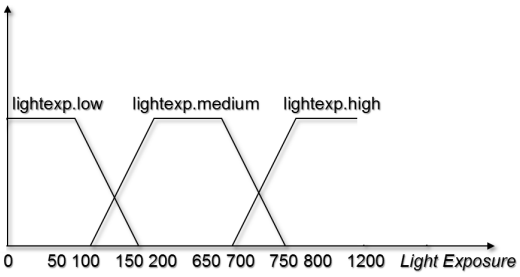


Figure 5.2: The executable code exchange mechanism allows to define fuzzy variables and their related membership functions on already deployed nodes. To remotely define the fuzzy variable `lightexp`, the code to be remotely executed must be enclosed between `tell:` and `:tell` and typed at the CLI of the bridge node that sends the executable code to the destination node. The remote node receives the sequence of words `0 1200 fvar lightexp` and locally interprets it.




```
0       1200    fvar    lightexp
0       0        50     150     member  lightexp.low
100     200     650     750     member  lightexp.medium
700     800     1200    1200    member  lightexp.high
```

Figure 5.3: Fuzzy sets associated with the fuzzy variable `lightexp`. On the right side, the code to define the fuzzy variable `lightexp` and its membership functions. The definition domain, corresponding to the raw readings values interval [0,1200], is given before the word `fvar`, while the word `member` defines each of the three trapezoidal membership functions by using four control points (bottom-left, top-left, top-right, and bottom-right).

can be easily defined on deployed nodes provided that the symbolic program is placed between `tell:` and `:tell` as indicated in Figure 5.2.

The representation of a fuzzy variable and its membership functions in memory is provided in Figure 5.4.

A node can be made measure light exposure, and fuzzify it with the code:

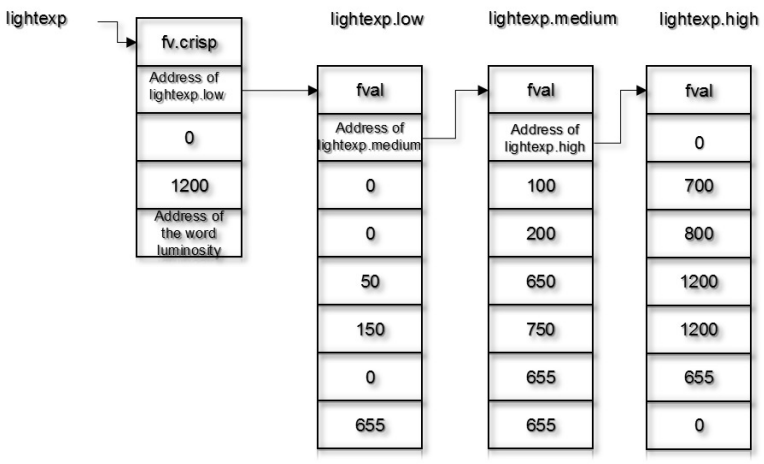


Figure 5.4: Memory representation of the fuzzy variable `lightexp` and its related membership functions after the code shown in Figure 5.3 is executed. The implementation refers to linked structures. Each link field stores the address of the next defined membership function. A link field that is equal to zero indicates the last membership function concerning that variable. It is worth noticing that the slope values are "scaled" to 65535 since this is the maximum number that can be expressed with 16-bits.

```
lightexp measure apply
```

The word `measure` fetches the xt field of the fuzzy variable that precedes it and executes the associated code. In detail, when the word `measure` is interpreted, the word address, which is stored in the xt field, is executed. Then, the word `luminosity` is executed and the sensory reading is left on top of the stack. This value is treated as crisp input by the word `apply`. As its name suggests, the word `apply` applies the crisp input to all the membership functions referring to `lightexp` and stores the fuzzy truth value in the correspondent `fval` field. Basically this word scans the linked list and fuzzifies the sensory reading for each membership function.

To access the truth value resulting from the fuzzification process the code:

```
lightexp.low @
```

pushes onto the stack the truth value by using the built-in word `@` (*fetch*). Rather than through a thresholding process, a device can establish if it is close to the

Table 5.1: Words defined in the dictionary to implement fuzzy reasoning according to (VanNorman, 1997).

Word	Description
`slope`	Compute the slope given two points of a side
`set-slope`	Set the left and right slope in the appropriate membership fields
`&`	Fuzzy AND
`|`	Fuzzy OR
∼	Fuzzy NOT
`=>`	Fuzzy implication
`fuzzify`	Given a crisp value and a membership, assign a membership value for it
`apply`	Apply the crisp input to the specified fuzzy input variable
`output`	Create an output fuzzy variable
`singleton`	Define a singleton output function
`rules`	Evaluate rules
`conclude`	Defuzzify and leave the crisp output on top of the stack

window through the evaluation of fuzzy rules in the form:

```
temp.high @ lightexp.high @ & => close-to-window
```

where `temp.high` and `lightexp.high` are membership functions of the fuzzy input variable `temp` and `lightexp` respectively, and `close-to-window` is one of the linguistic labels associated to the output variable. Similarly to the case of the thresholding process, if both the temperature and the light exposure levels are high a node can infer to be under sunlight, and thus close to the window.

5.1.2 Self-classification into Thermal Zones

Let us suppose the main goal is to make the deployed nodes able to discover their distribution with respect to thermal zones of an environment lighted by some windows exposed to direct sunlight, and lamps. Each node assesses in turn the thermal zone it belongs to, and makes the others aware of this information. The syntactic construct `classification` is defined to make the nodes able to classify according to an arbitrary number of fuzzy variables. With the previously defined input variables `temp` and `lightexp` the code:

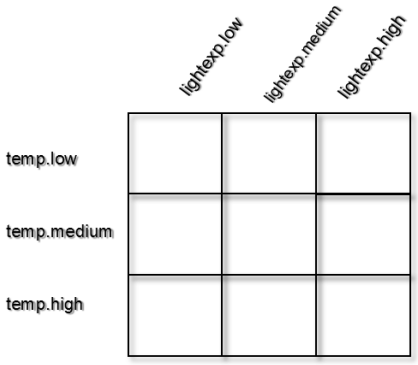


Figure 5.5: The execution of `temp lightexp 2 classification thermal-zone` creates the word `thermal-zone` that is bound to the two fuzzy variables. A 9 cells sized memory area is allocated as `temp` and `lightexp` have both three linguistic variables associated. In essence, each of these cells identifies a thermal-zone, a membership class according to which the node classifies itself. This area stores all the possible combinations for the rule evaluation process and aggregation.

```
temp lightexp 2 classification thermal-zone
```

creates the new word `thermal-zone`, which is bound to the two fuzzy variables `temp` and `lightexp` as illustrated in Figure 5.5.

When the new word `thermal-zone` is executed, it measures the temperature and luminosity, fuzzifies the crisp inputs and evaluates the rules by storing the firing strength for each rule, indicating the degree to which the rule matches the inputs. The rule generation process considers all the possible combinations of all the membership functions, -i.e. in this case, the set of all ordered pairs `(a,b)` where `a` and `b` are linguistic terms associated respectively with `temp` and `lightexp`. When handling few variables, this does not cause excessive memory occupation. It offers instead the advantage of considering a fine-grained classification based on all the n-tuples, that in this case, are all valid. However, optimization methods for the reduction of a large scale rule base may be required in real-time fuzzy systems (De Paola et al., 2014a; Jin, 2000; Yam et al., 1999). When needed, the table is traversed to compute the membership grade of the output by aggregating all rules. The rule with the maximum strength is taken as the output membership class (Figure 5.6). This way, each node is able to classify itself into one of the thermal zones. To support more sophisticated behaviors, it is possible to exploit

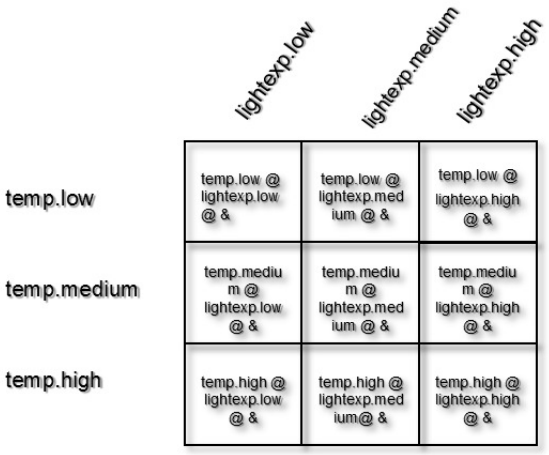


Figure 5.6: The rule generation involves the evaluation of all the possible combinations of the truth values of each membership function. Finally, the rule aggregation process consists in scanning the table to return the cell index storing the rule with the maximum strength. This index represents the class the node belongs to.

the mechanism of code exchange among nodes to trigger the process of neighbor discovery in order to keep track of their classification into thermal-zones.

For this purpose, it is necessary to define the table `nodes-distribution` to contain the number of nodes for each thermal zone (Figure 5.7). To start the whole classification process, the word `classification-start` can be sent to already deployed nodes through the executable code exchange paradigm. For instance, each device starts the timer and can transmit once, after waiting (word `on-timer`) for a time that is function of its unique ID. When the time elapses, the word `classification-spread` is executed, the node classifies itself into a thermal zone and then broadcasts the class it belongs to, together with the code to make the others update the whole distribution. The Forth code required for the entire process is the following:

```
: local-update
nodes-distribution update ;

: spread
dup local-update
```

```
bcst [tell:] ~ local-update [:tell] ;

: classification-spread
thermal-zone spread ;

: classification-start
start-timer
on-timer ['] classification-spread ;
```

in which the word `spread` creates a message with the code to make the other devices update locally the `nodes-distribution`. At the end of the `update` process, each node holds the current nodes distribution in terms of thermal zones, as such:

```
Class 1 2 3 4 5 6 7 8 9
    # 5 1 0 0 0 0 0 1 1
```

Five nodes belongs to class 1, one node to class 2 and so on. Each node knows the number of nodes in the network and their position, without any centralized computation. Once some nodes are moved from their position to another, and the process is triggered again, each node is able to detect the new distribution.

Moreover, the analysis of the nodes distribution may lead a node to classify itself as an outlier, to trigger self-diagnosis operations, and even to take specific actions, by reasoning about the whole network configuration and its membership thermal zone. The interactivity granted by the proposed approach permits the programmer to communicate with the network through the serial shell, i.e. the CLI, of the bridge node. For instance, the programmer can tell the nodes belonging to class 8 to turn their red LED on:

```
bcst tell: thermal-zone 8 class? [if]
   red led on [then] :tell ;
```

5.1.3 Experimental Results

Because of the limitations in terms of available resources, the implementation of symbolic reasoning on resource constrained devices must be particularly efficient.

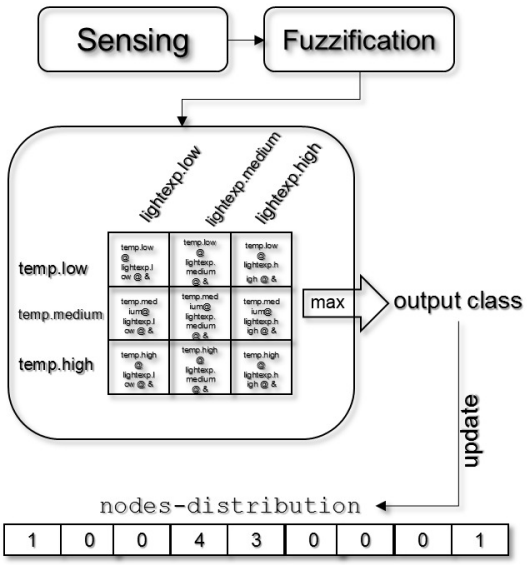


Figure 5.7: The word "thermal-zones" operates by sensing the temperature and light. Both sensory readings are treated as crisp input and fuzzified according to the membership functions of `lightexp` and `temp`. The rule generation considers all the pairs of truth values related to linguistic variables bound to different fuzzy variables. This is justified by the fact that each combination represents a different thermal zone identified by distinct temperature and light conditions. Indeed, the index of the cell storing the maximum value represents the thermal zone the node belongs to. The cell correspondent to the output class is incremented in nodes-distribution in order to allow each node to assess the distribution of the others.

This approach makes applications to be developed on real devices provided with a programming environment running directly on the target hardware. This prevents the presence of further intermediate layers between the hardware and software applications and increases efficiency. Moreover, as already discussed, although running very close to the hardware, symbolic computation allows to treat knowledge with a high degree of expressiveness.Distributed computation is less expensive than in mainstream approaches because executable code exchange is implemented at a very low level even though it exports a high-level interface to application code. The inclusion of reasoning mechanisms on resource constrained devices is particularly efficient as it occupies only 6 bytes of RAM and 863 bytes of Flash memory. The fuzzy wordset consists of 31 words. The application allowing the classification into thermal zones is quite compact since it consists of only 20 words

and occupies 560 bytes of RAM and 825 bytes of Flash.

5.2 An AmI application to control an HVAC system by estimating the user comfort level

Differently from the application previously described, in which symbolic reasoning is perfomed locally, this Section presents a Fuzzy Logic system where the value updates and the rule evaluations are performed in a distributed way. Through the proposed methodology, the development of an Ambient Intelligence application is discussed. In particular, it is described how the nodes may compute an estimation of the user thermal comfort by exchanging symbolic rather than numerical data, and control an HVAC (Heating, Ventilation and Air Conditioning) system accordingly.

5.2.1 Distributed Symbolic Reasoning with Fuzzy Logic

Integrating readings of all the nodes is one of the main tasks of applications running on resource-constrained devices, though processing is often carried out in a centralized manner. This Section alternatively proposes to make full use of the computing resources of nodes and implement a distributed application When information is expressed through crisp data it is possible to model the convergence to a single shared value through successive modification of the initial state toward the average of all the values held by the nodes:

$$x_t(t+1) = \sum_{j=1}^{N} W_{i,j} x_j(t). \tag{5.1}$$

where $W_{i,j}$ represents the weight related to the information held by the j-th node. Averaging is widely used in distributed WSNs applications to solve several problems, such as clock synchronization(Maggs et al., 2012). On the contrary, it is possible to achieve consensus among nodes through the exchange of symbolic and qualitative description of the observed phenomenon, as in the natural language (Mauris et al., 1994). There are many implementations of fuzzy controllers

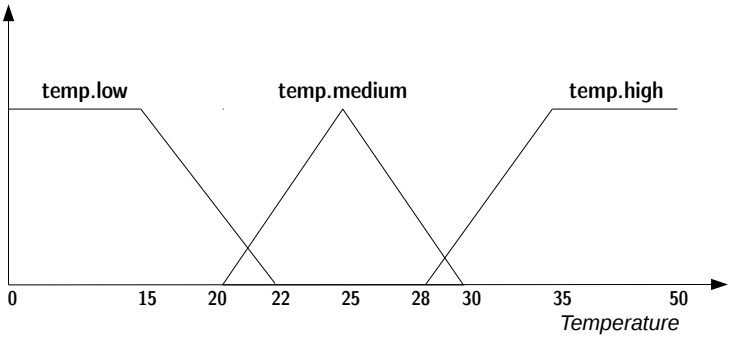


Figure 5.8: The three membership functions of the fuzzy variable `temp`. Four control points (bottom-left, top-left, top-right, and bottom-right) are used to define the shape of a membership function so that triangular and asymmetric trapezoidal shapes are defined by repeating one control point value as shown in Listing 5.1.

in literature with diverse formalisms. The formalism described in (VanNorman, 1997) has been extended to support both local and remote operations on fuzzy values.

Listing 5.1: Remote definition of the fuzzy variable `temp` and creation of the membership functions with the specified labels. In this example the message is broadcast using the special `bcst` address through a bridge node

```
1 bcst tell:
2   0 50 fvar temp
3   0 0 15 22 member temp.low
4  20 25 25 30 member temp.medium
5  28 35 50 50 member temp.high
6 :tell
```

Due to the hardware limitations of the microcontroller, fuzzy truth-values are represented with integer numbers. This is not a limiting factor, though as arbitrarily precise computations can be carried out with fixed point values adopting integer scaling factors. In this case, following the convention used in the original Fuzzy System implementation (VanNorman, 1997) the maximum fuzzy truth value is 255. By exploiting executable code exchange a fuzzy variable definition can be easily distributed among nodes even after their deployment. For instance, listing 5.1 reports the command line input that broadcast (`bcst`) the code to define the fuzzy

variable `temp` and its support (line 2) in the range [0,50] Celsius degrees, along with three trapezoidal membership functions (lines 3-5). Four control points are used to define the shape of a membership function (Fig. 5.8). After such definition all the nodes can evaluate, exchange and apply rules to fuzzy temperature values. For instance, a node can measure its local temperature and fuzzify it with the code:

```
temp-read temp apply
```

then make the others update their fuzzy temperature values accordingly:

```
bcst temp fvar-remote-update
```

The command evaluates the `temp` variable and broadcasts a message containing code that updates the three membership values. The word `fvar-remote-update` creates a message whose content is the repetition of the pattern

```
<truth> <membership func> fvar-update
```

for each membership function of the argument fuzzy variable– `temp` in this case. When a node receives the message, it interprets the command updating the truth values of its local `temp.low`, `temp.medium` and `temp.high` membership functions. Once the basics of fuzzy values exchange and evaluation are set, it is quite easy to use fuzzy symbol processing for distributed processing. Let us suppose it is required to build a thermostat application in which fuzzy temperature values from the network nodes are averaged and used as input to the control rules. A *fuzzy average* variable is then defined on all the nodes as:

```
bcst tell: temp favg avg-temp :tell
```

The word `favg` defines the new variable `avg-temp` binding it to the previously defined variable `temp` to store the fuzzy average temperature. The variable `avg-temp` then stores an average truth value for each membership function of the bound variable – `temp` in this case. The average truth value bound to a membership function is updated by stating its new value:

```
30 avg-temp temp.low favg-update
```

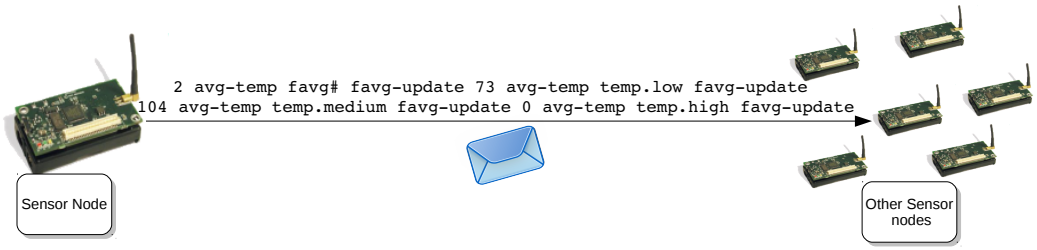


Figure 5.9: Executable code contained in the message sent by the word `favg-remote-update`. The word `favg-update` is used to update the number of samples (`favg#`) or the average truth value of a given membership function.

and the new number of samples of the average

```
1 avg-temp favg# favg-update
```

When the local update is completed it is propagated to the other nodes:

```
bcst avg-temp favg-remote-update
```

The word `favg-remote-update` builds a message containing the code to update the remote `favg` values as shown in Fig. 5.9. A summary of all the words implementing the formalism is provided in Table 3.1.

5.2.2 Control an HVAC system according to the user comfort level

In order to provide a practical application of the proposed methodology to a real problem, this Section focuses on an AmI scenario constituted by a sensory infrastructure that acquires temperature readings and reasons in a distributed way to control an HVAC system according to the user preferences (Fig. 5.10). The temperature sensing nodes are equipped with the MTS300CB sensing board, while the IR-receiver node and the IR-emitter node are provided with custom IR devices connected to the expansion lines of an MDA100CA prototyping board. The nodes are already calibrated and share the same fuzzy temperature categorization (low, medium, high) with the user. The fuzzy variables `temp`, `avg-temp`, along with the related fuzzy linguistic terms, are those discussed in section 5.2.1.

Detecting the user's comfort level

In order to represent the user thermal comfort, the fuzzy variable `user-comfort-level` and three membership functions (`ucl.cold`, `ucl.comfortable` and `ucl.hot`) are defined. These membership functions are known in advance and are the same for each node.

Listing 5.2: Words defined on the IR receiver node

```
1 : comfort-level-bcst ( -- )
2   bcst user-comfort-level
3   fvar-remote-update ;
4
5 : comfort-level-remote-update
6   translate-IR
7   comfort-level-bcst
8   bcst [tell:] user-agreement-mode
9    [:tell] ;
```

The IR receiver operation is described quite compactly by the code in Listing 5.2. The word `comfort-level-remote-update` implements the whole process described in Fig. 5.11. In the experimental setup the control command contains, among other codes related to the HVAC operating mode, a temperature setpoint value and the fan speed value. For the latter, only three values, low, medium and high, are possible. Thus, the word `translate-IR` (line 6) decodes the temperature setpoint and fan speed in the last received IR command. Then fuzzifies these crisp inputs and applies rules to obtain a crisp value representing the user input. Finally, it fuzzifies the user input value and stores the obtained fuzzy truth values for `ucl.cold`, `ucl.comfortable`, and `ucl.hot`. The word `comfort-level-bcst` broadcasts the updated value of the user comfort level using the word `fvar-remote-update` with the `bcst` MAC address. After the execution of `translate-IR` and `comfort-level-bcst`, the network is switched into the *user agreement mode* (Line 9), which is described in subsection 5.2.2. The mechanism through which the IR receiver node updates the *user comfort level* on the rest of the network is shown in Figure 5.12.

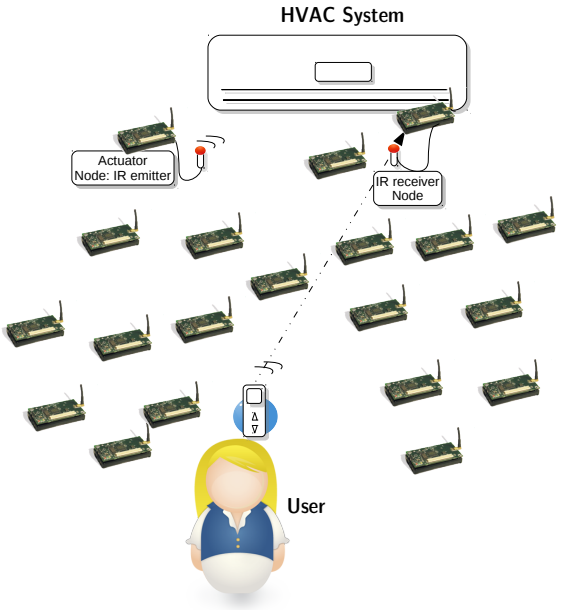


Figure 5.10: The AmI application scenario. The network is composed of temperature sensing nodes, a node able to detect the IR signals from the remote control to provide the network with *implicit user feedbacks*, and an actuator node equipped with an IR emitter to send commands to the HVAC system.

Agreement with the user

In order for the application to work properly it is necessary to provide each node with the capability to assess whether it has the same perception of the environmental conditions as the user. After the node has measured the temperature, the reading is fuzzified and processed by fuzzy rules together with the last user input received from the IR receiver and eventually defuzzified to obtain a crisp output representing the agreement with the user. The process is based on the mechanism of implicit feedbacks, as described in (De Paola et al., 2014b) and is shown in Fig. 5.13. After agreement with the user has been evaluated the network switches to the *network agreement mode*, described in subsection 5.2.2

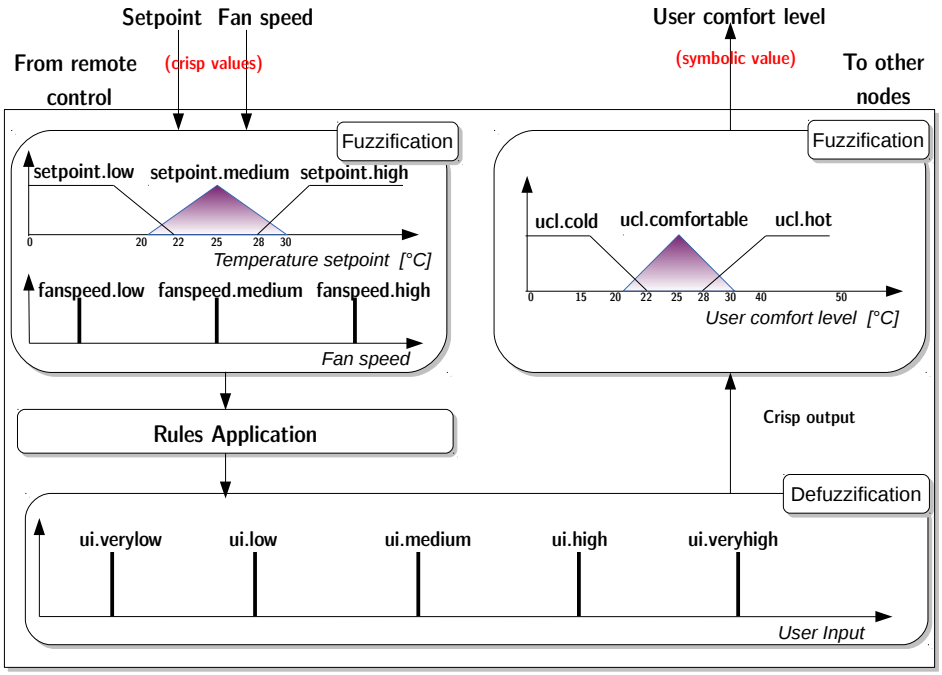


Figure 5.11: The IR receiver node processes the IR input signal to obtain the numerical values corresponding to the fan speed and the setpoint set by the user. These values are then fuzzified and combined into a representation of user input through fuzzy rules. Then the user-input is fuzzified and the IR receiver node computes a truth value for each membership function of the fuzzy variable `user-comfort-level`. Eventually, the symbolic values are broadcast to the WSN network.

Agreement with the network on temperature and comfort level estimation

Analogously to the agreement with the user, each node can assess its agreement degree with the temperature perceived by the network. Each node, before transmitting, locally updates the fuzzy average by adding its fuzzy truth value to the sum it has received, increments the number of nodes, calculates the average, and sends the sum of fuzzy truth values to the other nodes (Listing 5.3).

Listing 5.3: Code implementing the shared averaging of the fuzzy temperature value

```
1 : average&bcst
2   temp avg-temp favg-local-update
```

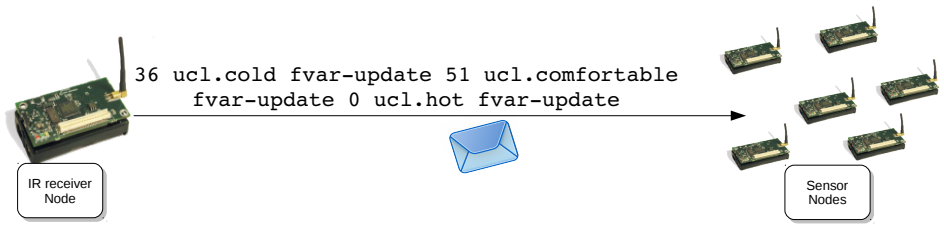


Figure 5.12: Remote update of the `fvar user-comfort-level`. The IR receiver node executes `comfort-level-bcst` once it detects an IR signal (Listing 5.2). This causes the execution of `fvar-remote-update`, which creates a message whose content is the repetition of the pattern: *truth-value linguistic-value* `fvar-update` for each membership function of the fuzzy variable. Once the message is received, the WSN node updates the fuzzy truth values related to the fuzzy values `ucl.cold`, `ucl.comfortable`, and `ucl.hot`.

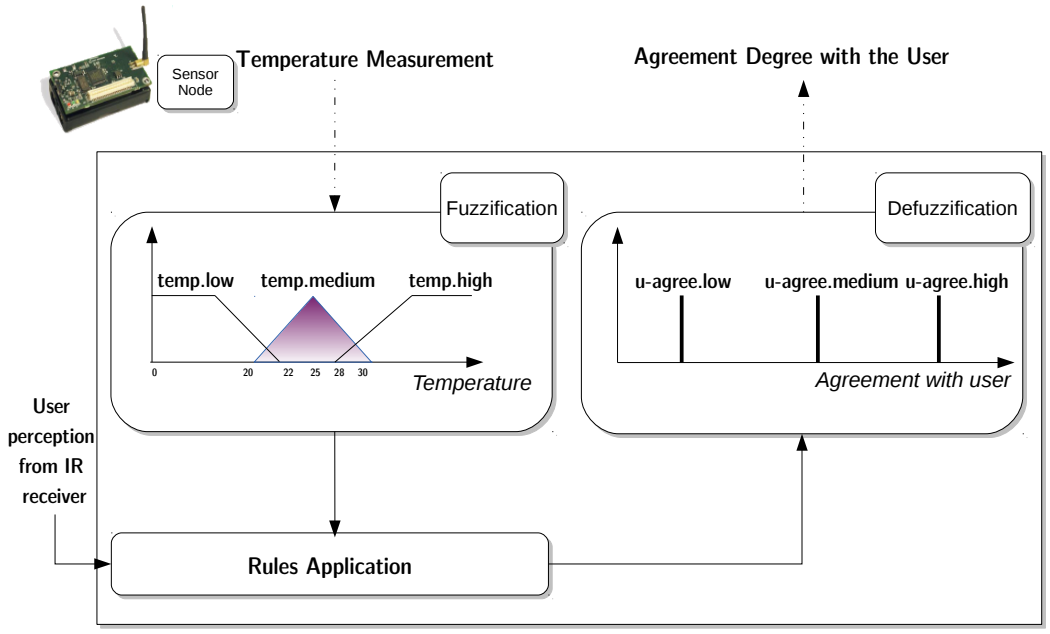


Figure 5.13: Description of the user-agreement mode. After performing the temperature measurement each node fuzzifies the crisp input and combines it with the user user comfort level estimate to obtain a crisp value representing the agreement degree with user. Fuzzy rules to evaluate the agreement with the user are described in Table 5.2.

```
3   bcst avg-temp favg-remote-update
4 ;
```

Table 5.2: Fuzzy rules used to evaluate the agreement with the user.

temp \ user-comfort-level	ucl.cold	ucl.comfortable	ucl.hot
temp.low	node-usr-agreement.high	node-usr-agreement.medium	node-usr-agreement.low
temp.medium	node-usr-agreement.medium	node-usr-agreement.high	node-usr-agreement.medium
temp.high	node-usr-agreement.low	node-usr-agreement.medium	node-usr-agreement.high

Table 5.3: Fuzzy rules used in the evaluation of the agreement with other nodes

temp \ avg-temp	temp.low	temp.medium	temp.hot
temp.low	nodes-agreement.high	nodes-agreement.medium	nodes-agreement.low
temp.medium	nodes-agreement.medium	nodes-agreement.high	nodes-agreement.medium
temp.high	nodes-agreement.low	nodes-agreement.medium	nodes-agreement.high

```
5 : start-averaging
6   id @ on-timer ['] average&bcst once fire ;
```

The word `favg-local-update` (line 2) updates the fuzzy average according to the values of `temp`. The word `favg-remote-update` sends the updating message to the other nodes. The word `start-averaging` controls the shared averaging process by making the word `average&bcst` be executed only once, after a time interval proportional to the node's ID value has elapsed. The word `network-agreement-assess` (Listing 5.4) implements the procedure to compute the agreement degree with the network. The temperature measurement is the crisp input to the fuzzification process. The resulting fuzzy values are combined with the user comfort level sent by the IR receiver and then defuzzified, obtaining a numerical value representing the agreement of the node with the rest of the network, as shown in Fig. 5.14. The word `network-agreement-rules` executes the rules described in Table 5.3 and then `conclude` defuzzifies the output fuzzy variable `network-agreement` leaving on the stack the resulting crisp output. The agreement with the other nodes can be used to define new behaviors according to the agreement degree. For instance, if the agreement degree is low, a node can decide to retract its contribution to the average and inform the network.

Listing 5.4: Code to assess agreement of node with the network

```
1 : network-agreement-assess ( -- crisp-output)
2   temp-read temp apply
```

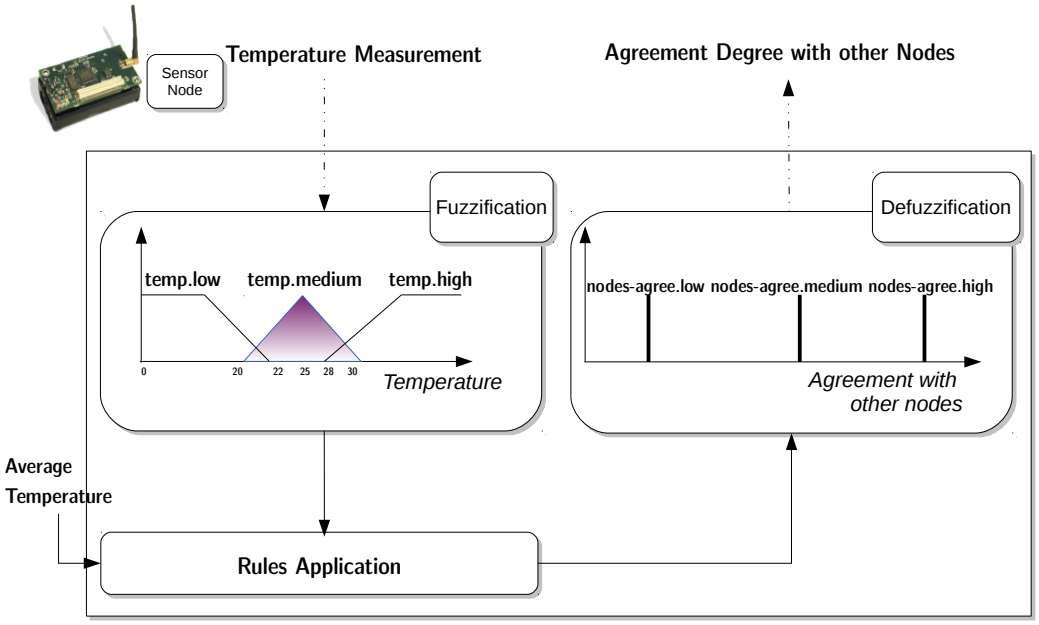


Figure 5.14: The network agreement task. Each node can evaluate its accordance with the network computed comfort level, that is the average temperature truth value resulting from the aggregation process. Nodes perform the fuzzification-rules-defuzzification loop to obtain a crisp output representing its agreement degree with the rest of the network.

```
3   network-agreement-rules
4   network-agreement conclude
```

Actuate on the environment

In order to meet the user's thermal comfort, it is necessary a periodical control of the HVAC system. Since the membership functions representing the user comfort level are known to the nodes in advance, the average temperature classification can be combined with the former to control the HVAC system. Output values are the corrections in fan speed and temperature setpoint to be sent to the HVAC as IR signals. Before sending an IR command, the actuator sends to the IR-receiver node a command to disable IR detection and processing. This ensures that only IR signals coming from the user operated remote are interpreted as user inputs. After the signal is sent the actuator re-enables IR signal reception.

5.2.3 Building and testing the application

The proposed methodology supports and encourages interactive and incremental programming. Following this coding practice, the steps it takes to build and test the experimental AmI application are described. All nodes receive the same code –with the few exceptions of the specific code for the IR receiver and transmitter nodes– in the initial *code distribution* phase that can be carried on in either batch or interactive way. In the latter case, the programmer sends nodes definitions for:

- the fuzzy variable `temp` and the related membership functions (Listing 5.1),
- the fuzzy average variable `avg-temp`,
- the action to be taken when a IR signal is detected (Listing 5.2),
- the shared averaging of the fuzzy temperature value (Listing 5.3),
- the assessment of the agreement of node with the network (Listing 5.4).

When all the necessary definitions have been provided to nodes, the fuzzy temperature averaging can be started:

```
bcst tell: start-averaging :tell
```

and the handler responsible for the IR signal interpretation can be installed on the IR receiver node:

```
<IRreceiverID> tell: on-IRreception
' comfort-level-remote-update :tell
```

Periodically, the node with the IR emitter might choose one of its neighbor WSN nodes to ask the network estimated comfort level resulting from the averaging process, and use it to actuate on the environment. The control time interval must be adequate to ensure that the nodes have already calculated the degree of agreement with the network at least once. A comprehensive description of the concurrent application execution is provided in Figure 5.15. The *inner interpreter loop* executes the commands received from either a UART or a radio link. As the execution model is event-driven, other tasks can be running on nodes and paused when the tasks of the application are to be executed. This mechanism is quite effective as it is based on hardware interrupts.

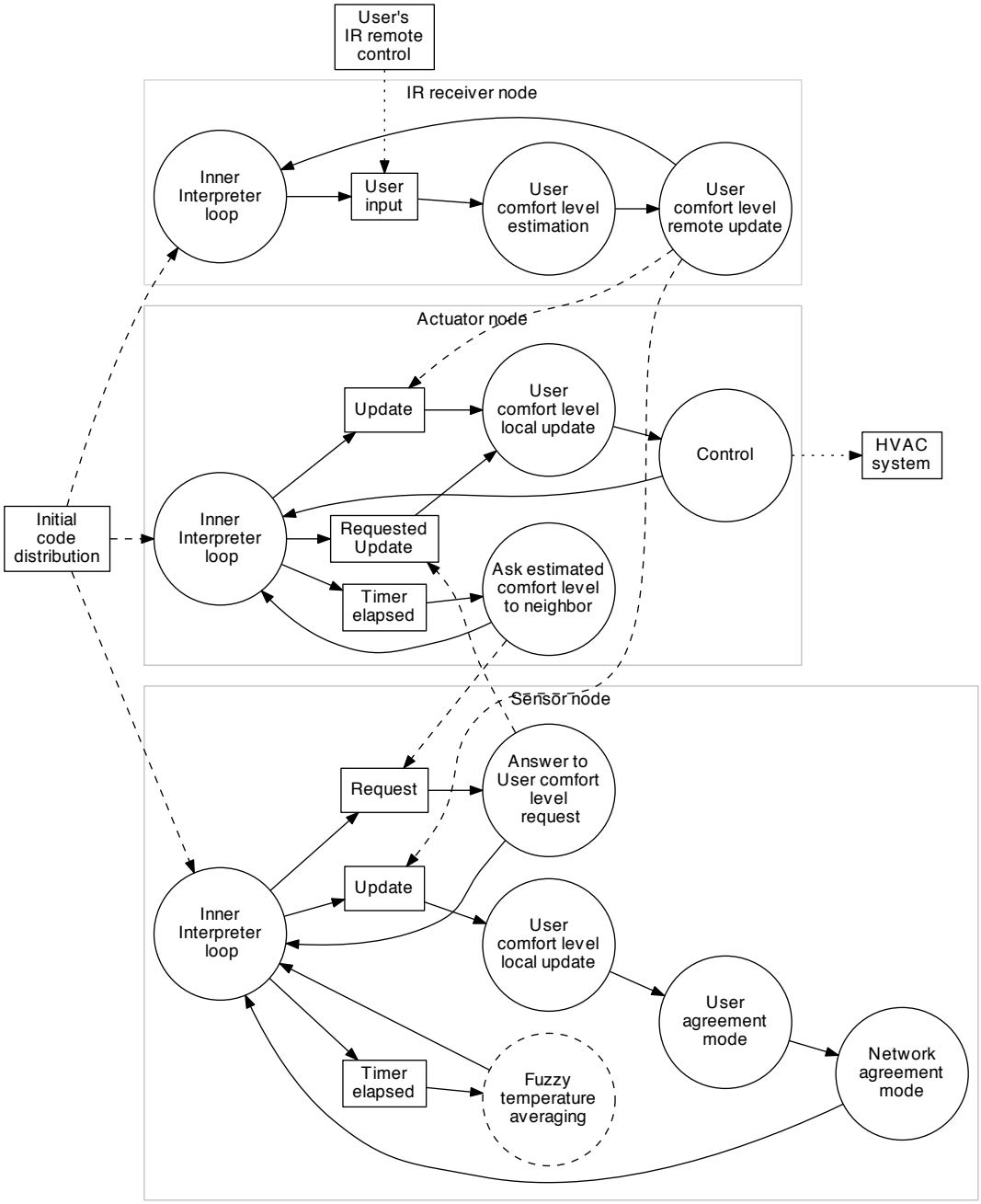


Figure 5.15: Concurrent execution of the AmI distributed application. Continuous arrows indicate execution of the task composing the application, which are drawn in circles. Signals to trigger task executions –in boxes– are represented by dashed arrows. These signals are actually messages containing executable code that are sent between nodes. The *Fuzzy temperature averaging* task is enclosed in a dashed circle to denote its distributed execution, as described in subsection 5.2.2 (Listing 5.3). The dotted arrows from the *IR Remote Control* and to the *HVAC System* represent remote control IR signals.

5.3 A Remote Shell Application

This section describes a simple application to demonstrate the simplicity of the proposed development methodology and the compactness of the resulting code. The application is a telnet-like remote shell that allows to transparently interact with the shell of the bridge node as if it were the shell of a remote node, either sensor or actuator.

Building applications in an interactive way reduces the development time and allows to simultaneously test the application.

The experimental setup includes deployed sensors and actuators nodes, and a bridge node connected to the user computer. The user interacts directly with the shell of this node as a means to send both data and code to be remotely executed.

A remote shell application can be useful in different contexts, for instance for debugging operations, to inspect the state of a remote node or to display the sensor readings as if the remote node were physically connected to the programmer console. During code drafting, the programmer can interactively test node operation and verify that the application is working properly.

What it is discussed in this section is a coding example with the main purpose to show the readability and expressiveness of the code and the simplicity of development.The code is fully functional and has been extensively used in experimentations. Indeed, with hindsight, the remote shell application can be considered as a completeness and consistency test for the proposed approach. In fact, to implement this essential application, the words needed by the application have been defined interactively and tested during the development phase. Summarizing, it was necessary to define:

- words to redirect the output to the outgoing message;
- words for substituting code at runtime;
- syntactic structures to send data and code in a symbolic way;
- words to redirect the input to the radio and inject the ingoing message into the inner interpret input buffer.

The almost complete remote shell application code is shown in Listing 5.5. A few additional words are omitted and their description can be found in Table 5.4.

Listing 5.5: Code for a simple remote shell application

```
80 constant cmd-maxlen
variable cmd cmd-maxlen cells allot
variable cmd-len
variable node_id
variable timeout

: input-send ( -- )
   cmd cmd-len @
   node_id @ [tell:] ~s [:tell] ;

: rshell-task (  -- )
   payld-reset
   input-send
   timeout @ wait-answer if
   payld-print then ;

: user-input ( -- )
   cmd cmd-maxlen accept ( -- len )
   cmd-len ! ;

: close ( -- )
   node_id @ [tell:] -radio-output
   [:tell] quit ;

: rshell-loop ( -- )
   begin
    cr ." rsh>" user-input
    close?
    if close
    else rshell-task
    then
```

```
32      again ;
33
34  : on-timeout ( -- )
35      ." Connection timeout." cr ;
36
37  : welcome-msg
38      ." Welcome to the remote shell
39   application!" cr
40      ." Enter 'close' to close the
41   application" cr ;
42
43  : rshell ( id -- )
44      welcome-msg
45      2000 timeout  !
46      dup node_id !
47      [tell:] +radio-output [:tell]
48      rshell-loop  ;
```

The core word of this telnet-like application is `rshell-loop`. It consists in an endless loop that displays the remote shell prompt (line 27) and waits for the input code from the user. If the user types `close` the application quits, otherwise the `rshell-task` word is performed. This word sends the user input to the remote node (line 13). The bridge node waits for input from the radio, i.e. incoming frames from the destination node (line 14), for at most the amount of time specified by the `timeout` variable, which defaults to 2000 ms (line 45). This wait phase is implemented as a loop with timeout. If the timeout expires without receiving an answer from the remote node, a timeout exception handled by `on-timeout` occurs, otherwise the bridge node displays the incoming frame content, i.e. its payload (line 15).

To start a remote shell session with the node named *node1*, using a telnet style convention the user types:

```
node1 rshell
```

and an initial welcome message is displayed (line 44). The word `rshell` stores *node1*'s address in the variable node_id (line 46) and sends a message to the

Table 5.4: Summary table of additional words used in the remote shell application

Word (before -- after)	**Description**
+radio-output (--)	Redirect the output to the radio. This word is part of the Radio I/O word set
-radio-output (--)	Redirect the output to the UART. This word is part of the Radio I/O word set
radio-input? (timeout -- flag)	Check for incoming radio messages. If no message arrives before `timeout` milliseconds leave false on the stack, otherwise leave true
payld-reset (--)	Set to 0 the incoming payload length and its current pointer
payld-print (--)	Display the incoming frame content (i.e. the payload)
wait-answer (timeout -- flag)	Wait for incoming radio frame for a predefined period of time specified by the `timeout` variable. If the timeout expires without receiving any answer message, an exception handled by `on-timeout` occurs
user-input (--)	Wait for user input and store its content in the `cmd` buffer and its length in the `cmd-len` variable for further processing by `close?` and `input-send`

remote node to switch its output device to the outgoing message (line 47). After that, it performs the remote shell main loop (line 48).

As a result, the user interacts with the remote node as if the remote node shell were physically connected to the user computer. Table 5.4 summarizes the additional words used in the the remote shell application code.

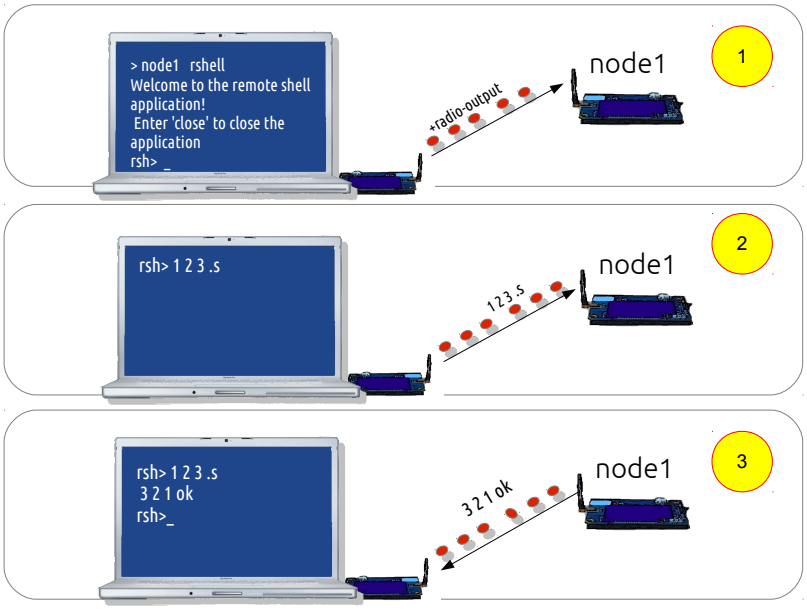


Figure 5.16: Key steps of the remote shell application execution. First, the welcome message is displayed together with the remote shell prompt. A message including the code to redirect the remote node output to the outgoing messages is sent to the remote destination node (1). The user pushes some numbers to the remote data stack and commands to display the stack content (2). The remote stack is displayed in the terminal (3). The interaction takes place as the remote node were physically connected with the user computer.

Chapter 6

Conclusions

The design and verification of CPSs are challenging tasks that require the programmer to combine models and languages to treat different domain aspects while dealing with resource constraints. Novel programming paradigms and high-level abstractions are needed to address these issues and to support interoperability in AmI and IoT scenarios.

This work proposes a development methodology for distributed computing based on a very high-level programming paradigm that can be used with CPS resource-constrained devices such as those found in WSNs. Through a symbolic formalism, the executable code related to task description is modeled as close as possible to the respective functional specifications provided in natural language. A knowledge-based system for the development of CPS applications as sequences of words matching natural language patterns was devised. The proposed approach exploits a knowledge base integrating specifications about the Cyber-Physical domain, hardware, and programming patterns. The knowledge base explicitly defines the mapping between recurring CPS operations, such as sensing and actuation, and the corresponding hardware. Using the knowledge base, the system can generate oracles and test cases for runtime verification of CPSs on the target hardware. Stress test code can be also produced by the system to verify that program components behave as expected in repeated execution.

The proposed methodology permits to develop meaningful implementations that are strictly coupled to the specifications and eases detecting operational and

functional faults during the task design.

As an example, it has been shown how to proceed from the descriptions provided by data-sheets to a high-level implementation of a driver for a WSN radio transceiver chip. The resulting executable specification also provides an oracle for the runtime verification of the hardware module. The final code is quite understandable and coherent with the FSM functional model in the data-sheets.

Finally, Distributed Computing for Constrained Devices (DC4CD), a novel software architecture was introduced. DC4CD was envisioned and then developed specifically to support the experimentation on symbolic distributed programming of CPSs that include devices with limited resources. An important class of these devices is that of WSN nodes, whose extremely limited resources make implementing cooperative behaviors very difficult. Differently from other approaches proposed in literature, DC4CD is based on a computational paradigm that integrates the functionalities of a high-level symbolic interpreter, a compiler, and an operating system. Symbolic code is exchanged by nodes as pure text strings, with no loss of expressiveness and without requiring predefined message formats. In spite of this very high-level approach to code exchange, DC4CD compares favorably with the other available software architectures for symbolic processing, in both performance and code compactness, while also providing more features such as wireless reprogramming of deployed nodes, and interrupts and hardware handling at symbolic code level.

From a high-level perspective, future work may extend the rule-based system to enable system-level verification of distributed CPSs. Physical world concepts, abstract models, time, hardware components, communication schemes, and distributed high-level implementations can be integrated to generate oracles, in the form of executable code, to be sent to remote devices. The oracle could be exploited to (i) complement the information provided by datasheets (ii) verify that internal hardware states have been reached by target devices (iii) gather useful metrics, e.g. the time needed for message propagation or to accomplish a distributed task. From a low-level perspective, security aspects and authentication mechanisms for hardware protection, e.g. permanent storage writing, may be investigated. Compiler optimization may be also addressed in order to improve the platform performance.

APPENDIX

Fundamentals of Forth

Since its first appearance in 1972, Forth has been strongly characterized as a development tool to exploit efficiently the hardware, avoiding the difficulties of assembly coding (Ertl, 2011; Read, 2014). Rather than as a mere programming language, Forth is better defined as a whole made up also by a software design methodology, a development environment, an interactive interpreter, a portable compiler (Stoddart et al., 2012; Ertl, 2013), and an operating system. In spite of this multifaceted nature a complete Forth system can be implemented very compactly, as shown in Figure 4.2.

Forth has a very simple grammar that gives ample freedom to the programmer. Indeed, Forth fosters the design of application-specific languages.

A Forth program is simply a sequence of high-level symbols, called *words*, defined in the *dictionary*, and literal constants.

A Forth environment provides a command line interface (CLI) allowing for interactive development. Executing a word can be as simple as typing its name in a terminal. The interactive interpreter evaluates input by looking for words in the dictionary and executing their definition. The dictionary can be seen as a linked list in which each stored word maintains a pointer to the previous defined word.

A dictionary entry stores the word name and some useful information, including the code to be executed when the word is interpreted by the system, which is called "execution token"– or briefly *xt*. New words can be defined as sequences of already defined or built-in words.

Forth is based on the explicit management of two LIFO stacks, the *data stack*

and the *return stack*. Words use the data stack to pass parameters to each other. After a word has been executed, the result of its execution is pushed onto the stack and similarly, before its execution, the required parameters are pulled from the stack.

Since words may directly affect the stack, Forth programs often contain comments enclosed between parenthesis, known as *stack effect* notation, to represent the state of the stack before the word is executed and its state afterwards. The parameters required by the word precede the dashes, those left on the stack follow the dashes. The top of the stack before and after a word execution is the rightmost symbol on both sides. The stack notation of the word `+`:

```
( n1 n2 - n1+n2 )
```

shows that two operands are popped from the stack and then their sum is pushed back on the stack. The evaluation mechanism follows the Reverse Polish Notation (RPN).

The return stack is used to hold return addresses for nested definitions or to store temporary data. Both stacks are directly controlled by the programmer, who has full control of the system.

Forth systems include a number of words for stack manipulation, memory operations, I/O, interrupt management and so on.

Forth types comprise the *cell*, which is usually as large as the host CPU registers [1], and the *double cell*. Values of these types can be used in push and pop stack operations, as well as in and *fetch* (`@`) and *store* (`!`) memory operations. Single byte (*character*) values can also be used in *fetch* (`C@`) and *store* (`C!`) memory operations.

The main strength of Forth stands in the combination of interactivity, interpretation and compilation in the same programming environment.

The execution of the word `:` (colon) makes the system enter the compilation mode. A new entry is added to the dictionary, called as the word that follows the colon. Then the compiler parses the rest of the definition. Each following word is looked for into the dictionary and its *xt* compiled in the code section of the entry.

[1] In AmForth the *cell* size is 16-bit as, even if target AVR MCU is provided with 8-bit data registers, arithmetic and logic instructions can operate on 16-bit register pairs.

Constant values can be included in the definition and are compiled as *literals*, i.e the *xt* of some value-specifc word is added to the definition along with the value. The exact coding of the definition depends on the chosen execution model (cf. Section 4.2). When the word `;` (semi-colon) is encountered the system re-enters the interpreter mode. For instance, the following code defines a new word `++` that applies the sum operator twice:

```
:  ++ ( a b c - b+c+a ) + + ;
```

In this case the defined word has only a run-time behavior, however it is possible to define also *compiling* and *defining* words – i.e. words that have both a run-time behavior and a compile-time behavior– similarly.

Words can also have a deferred implementation. These words are executed indirectly through a vector containing an xt that points to the word to be actually executed. This technique, called "vectored execution", allows to change the behavior of these words at runtime. Deferred words `preamble` and `conclusion` have been used in Chapter 3 to switch on/off the runtime component verification tool. For instance, the code:

```
' noop is preamble
' noop is conclusion
```

sets the words `preamble` and `conclusion` to `noop` operation. The execution of the word `'` leaves on the stack the xt of the next word. The word `is` effectively sets the xt of the following word,– e.g. `preamble` –, to the xt on top of the stack,– i.e. in this case, the xt of `noop`.

Bibliography

AmForth Documentation, 2013. Available online at `http://amforth.sourceforge.net/amforth.pdf`.

Damián Adalid, Alberto Salmerón, María del Mar Gallardo, and Pedro Merino. Using SPIN for automated debugging of infinite executions of java programs. *Journal of Systems and Software*, 90:61 – 75, 2014. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2013.10.056. URL `http://www.sciencedirect.com/science/article/pii/S0164121213002641`.

I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393 – 422, 2002.

D. Alessandrelli, M. Petracca, and P. Pagano. T-Res: Enabling Reconfigurable In-network Processing in IoT-based WSNs. In *Distributed Computing in Sensor Systems (DCOSS), 2013 IEEE International Conference on*, pages 337–344, May 2013. doi: 10.1109/DCOSS.2013.75.

C. Alippi, R. Camplani, M. Roveri, and L. Vaccaro. REEL: A Real-time, Computationally-efficient, Reprogrammable Framework for Wireless Sensor Networks. In *Sensors, 2011 IEEE*, pages 1193–1196, Oct 2011. doi: 10.1109/ICSENS.2011.6126919.

A. Antola, A.L. Mezzalira, and M. Roveri. GINGER: a Minimizing-effects Reprogramming Paradigm for Distributed Sensor Networks. In *Robotic and Sensors Environments (ROSE), 2014 IEEE International Symposium on*, pages 88–93, Oct 2014. doi: 10.1109/ROSE.2014.6952989.

W. Araujo, L. C. Briand, and Y. Labiche. Enabling the runtime assertion checking of concurrent contracts for the Java modeling language. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 786–795, May 2011. doi: 10.1145/1985793.1985903.

Faisal Aslam, Luminous Fennell, Christian Schindelhauer, Peter Thiemann, Gidon Ernst, Elmar Haussmann, Stefan RÃ¼hrup, and ZastashA. Uzmi. Optimized Java Binary and Virtual Machine for Tiny Motes. In Rajmohan Rajaraman, Thomas Moscibroda, Adam Dunkels, and Anna Scaglione, editors, *Distributed Computing in Sensor Systems*, volume 6131 of *Lecture Notes in Computer Science*, pages 15–30. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-13650-4. doi: 10.1007/978-3-642-13651-1_2.

Atmel-ATRF230. AT86RF230 Datasheet, 2016. Available online at `http://www.atmel.com/images/doc5131.pdf`.

Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Computer Networks*, 54(15):2787 – 2805, 2010. ISSN 1389-1286. doi: http://dx.doi.org/10.1016/j.comnet.2010.05.010.

E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41 (5):507–525, May 2015. ISSN 0098-5589. doi: 10.1109/TSE.2014.2372785.

Stefano Bocchino, Szymon Fedor, and Matteo Petracca. PyFUNS: A Python Framework for Ubiquitous Networked Sensors. In *Wireless Sensor Networks*, pages 1–18. Springer, 2015.

Niels Brouwers, Koen Langendoen, and Peter Corke. Darjeeling, a feature-rich VM for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 169–182, New York, NY, USA, 2009a. ACM. ISBN 978-1-60558-519-2. doi: 10.1145/1644038.1644056. URL `http://doi.acm.org/10.1145/1644038.1644056`.

Niels Brouwers, Koen Langendoen, and Peter Corke. Darjeeling, a feature-rich VM for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 169–182. ACM, 2009b.

Doina Bucur, Giovanni Iacca, Giovanni Squillero, and Alberto Tonda. The Tradeoffs Between Data Delivery Ratio and Energy Costs in Wireless Sensor Networks: A Multi-objectiveevolutionary Framework for Protocol Analysis. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 1071–1078, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2662-9. doi: 10.1145/2576768.2598384. URL `http://doi.acm.org/10.1145/2576768.2598384`.

J. Cecilio and P. Furtado. Architecture for Uniform (Re)Configuration and Processing Over Embedded Sensor and Actuator Networks. *Industrial Informatics, IEEE Transactions on*, 10(1):53–60, Feb 2014. ISSN 1551-3203. doi: 10.1109/TII.2013.2262944.

Ioannis Chatzigiannakis, Andrea Vitaletti, and Apostolos Pyrgelis. A privacy-preserving smart parking system using an iot elliptic curve based security platform. *Computer Communications*, pages –, 2016. ISSN 0140-3664. doi: http://dx.doi.org/10.1016/j.comcom.2016.03.014. URL `http://www.sciencedirect.com/science/article/pii/S014036641630072X`.

Rui Chu, Lin Gu, Yunhao Liu, Mo Li, and Xicheng Lu. SenSmart: Adaptive Stack Management for Multitasking Sensor Networks. *Computers, IEEE Transactions on*, 62(1):137–150, Jan 2013. ISSN 0018-9340. doi: 10.1109/TC.2011.238.

Alessandra De Paola, Giuseppe Lo Re, and Antonio Pellegrino. A Fuzzy Adaptive Controller for an Ambient Intelligence Scenario. In Salvatore Gaglio and Giuseppe Lo Re, editors, *Advances onto the Internet of Things*, volume 260 of *Advances in Intelligent Systems and Computing*, pages 47–59. Springer International Publishing, 2014a.

Alessandra De Paola, Marco Ortolani, Giuseppe Lo Re, Giuseppe Anastasi, and Sajal K. Das. Intelligent Management Systems for Energy Efficiency in Buildings: A Survey. *ACM Comput. Surv.*, 47(1):13:1–13:38, June 2014b.

Xi Deng and Yuanyuan Yang. Online Adaptive Compression in Delay Sensitive Wireless Sensor Networks. *Computers, IEEE Transactions on*, 61(10):1429–1442, Oct 2012. ISSN 0018-9340. doi: 10.1109/TC.2011.174.

P. Derler, E. A. Lee, and A. Sangiovanni Vincentelli. Modeling Cyber-Physical Systems. *Proceedings of the IEEE*, 100(1):13–28, Jan 2012. ISSN 0018-9219. doi: 10.1109/JPROC.2011.2160929.

Wei Dong, Chun Chen, Xue Liu, Yunhao Liu, Jiajun Bu, and Kougen Zheng. SenSpire OS: A Predictable, Flexible, and Efficient Operating System for Wireless Sensor Networks. *Computers, IEEE Transactions on*, 60(12):1788–1801, Dec 2011. ISSN 0018-9340. doi: 10.1109/TC.2011.58.

Wei Dong, Yunhao Liu, Chun Chen, Jiajun Bu, Chao Huang, and Zhiwei Zhao. R2: Incremental Reprogramming Using Relocatable Code in Networked Embedded Systems. *Computers, IEEE Transactions on*, 62(9):1837–1849, Sept 2013. ISSN 0018-9340. doi: 10.1109/TC.2012.161.

Wei Dong, Yunhao Liu, Chun Chen, Lin Gu, and Xiaofan Wu. Elon: Enabling Efficient and Long-term Reprogramming for Wireless Sensor Networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4):77, 2014.

Adam Dunkels. A Low-Overhead Script Language for Tiny Networked Embedded Systems. Technical report, Swedish Institute of Computer Science, 2006.

Adam Dunkels, Joakim Eriksson, Niclas Finne, and Nicolas Tsiftes. Powertrace: Network-level Power Profiling for Low-power Wireless Networks. Technical report, Swedish Institute of Computer Science, 2011.

M Anton Ertl. Threaded code variations and optimizations. In *EuroForth 2001 Conference Proceedings*, pages 49–55. Citeseer, 2001.

M Anton Ertl. Ways to Reduce the Stack Depth. In *27th EuroForth Conference*, pages 36–41, 2011.

M Anton Ertl. PAF: A Portable Assembly Language. In *29th EuroForth Conference*, pages 30–38, 2013.

E. Estevez and M. Marcos. Model-Based Validation of Industrial Control Systems. *IEEE Transactions on Industrial Informatics*, 8(2):302–310, May 2012. ISSN 1551-3203. doi: 10.1109/TII.2011.2174248.

O. Evangelatos, K. Samarasinghe, and J. Rolim. Evaluating Design Approaches for Smart Building Systems. In *2012 IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012)*, volume Supplement, pages 1–7, Oct 2012. doi: 10.1109/MASS.2012.6708524.

L. Evers, P. Havinga, and J. Kuper. Dynamic Sensor Network Reprogramming using Sensorscheme. In *2007 IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–5, Sept 2007a. doi: 10.1109/PIMRC.2007.4394138.

L. Evers, P.J.M. Havinga, J. Kuper, M.E.M. Lijding, and N. Meratnia. Sensorscheme: Supply chain management automation using wireless sensor networks. In *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, pages 448–455, Sept 2007b. doi: 10.1109/EFTA.2007.4416802.

Muhammad Omer Farooq and Thomas Kunz. Operating Systems for Wireless Sensor Networks: A Survey. *Sensors*, 11(6):5900–5930, 2011.

S. Fischmeister and P. Lam. Time-Aware Instrumentation of Embedded Software. *IEEE Transactions on Industrial Informatics*, 6(4):652–663, Nov 2010. ISSN 1551-3203. doi: 10.1109/TII.2010.2068304.

Carolina Fortuna and Mihael Mohorcic. A Framework for Dynamic Composition of Communication Services. *ACM Trans. Sen. Netw.*, 11(2):32:1–32:43, December 2014. ISSN 1550-4859. doi: 10.1145/2678216. URL `http://doi.acm.org/10.1145/2678216`.

S. Gaglio, G. L. Re, G. Martorella, and D. Peri. A Symbolic Distributed Event Detection Scheme for Wireless Sensor Networks. *Proceedings of the 2016 IEEE Emerging Technology and Factory Automation (ETFA), to appear*, pages –, 2016.

Salvatore Gaglio, Giuseppe Lo Re, Gloria Martorella, and Daniele Peri. A Fast and Interactive Approach to Application Development on Wireless Sensor and Actuator Networks. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, pages 1–8, Sept 2014. doi: 10.1109/ETFA.2014.7005179.

Sudipto Ghosh and John L. Kelly. Bytecode fault injection for java software. *Journal of Systems and Software*, 81(11):2034 – 2043, 2008. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2008.02.047. URL `http://www.sciencedirect.com/science/article/pii/S0164121208000484`.

Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, Maria Kazandjieva, David Moss, and Philip Levis. CTP: An Efficient, Robust, and Reliable Collection Tree Protocol for Wireless Sensor Networks. *ACM Trans. Sen. Netw.*, 10(1): 16:1–16:49, December 2013. ISSN 1550-4859. doi: 10.1145/2529988. URL `http://doi.acm.org/10.1145/2529988`.

Bin Guo, Daqing Zhang, Zhiwen Yu, Yunji Liang, Zhu Wang, and Xingshe Zhou. From the Internet of Things to Embedded Intelligence. *World Wide Web*, 16(4): 399–420, 2013. ISSN 1386-145X. doi: 10.1007/s11280-012-0188-y.

H. F. Guo, L. Cao, Y. Song, and Z. Qiu. Automated Test Oracle Generation via Denotational Semantics. In *2014 14th International Conference on Quality Software*, pages 139–144, Oct 2014. doi: 10.1109/QSIC.2014.38.

Hai-Feng Guo. A semantic approach for automated test oracle generation. *Computer Languages, Systems & Structures*, 45:204 – 219, 2016. ISSN 1477-8424. doi: http://dx.doi.org/10.1016/j.cl.2016.01.006. URL `http://www.sciencedirect.com/science/article/pii/S147784241530021X`.

Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. A comprehensive survey of trends in oracles for software testing, 2013.

W. Hasanain, Y. Labiche, and S. Gheorghe. Automated State-based Online Testing Real-time Embedded Software with RTEdge. In *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*, pages 294–302, Feb 2015.

Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel Briand. Environment Modeling and Simulation for Automated Testing of Soft Real-time Embedded Software. *Software & Systems Modeling*, 14(1):483–524, 2015.

P. Iyenghar, C. Westerkamp, J. Wuebbelmann, and E. Pulvermueller. An architecture for deploying model based testing in embedded systems. In *Specification Design Languages (FDL 2010), 2010 Forum on*, pages 1–6, Sept 2010. doi: 10.1049/ic.2010.0153.

P. Iyenghar, E. Pulvermueller, M. Spieker, J. Wuebbelmann, and C. Westerkamp. Time and memory-aware runtime monitoring for executing model-based test cases in embedded systems. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, pages 506–512, July 2013. doi: 10.1109/INDIN.2013.6622936.

Yaochu Jin. Fuzzy Modeling of High-dimensional Systems: Complexity Reduction and Interpretability Improvement. *Fuzzy Systems, IEEE Transactions on*, 8(2): 212–221, Apr 2000.

S.K. Khaitan and J.D. McCalley. Design Techniques and Applications of Cyberphysical Systems: A Survey. *Systems Journal, IEEE*, 9(2):350–365, June 2015. ISSN 1932-8184. doi: 10.1109/JSYST.2014.2322503.

G. Kortuem, F. Kawsar, D. Fitton, and V. Sundramoorthy. Smart Objects as Building Blocks for the Internet of Things. *Internet Computing, IEEE*, 14(1): 44–51, Jan 2010. ISSN 1089-7801. doi: 10.1109/MIC.2009.143.

M. Kovatsch, S. Mayer, and B. Ostermaier. Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 751–756, July 2012. doi: 10.1109/IMIS.2012.104.

Richard Lai and Ajin Jirachiefpattana. *Communication Protocol Specification and Verification*. Springer Publishing Company, Incorporated, 2013.

Helen C Leligou, Christos Massouros, Eleftherios Tsampasis, Theodore Zahariadis, Dimitrios Bargiotas, Konstantinos Papadopoulos, and Stamatis Voliotis. Reprogramming Wireless Sensor Nodes. *International Journal of Computer Trends and Technology, May to June issue 2011*, 7, 2011.

Philip Levis and David Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *ACM Sigplan Notices*, volume 37, pages 85–95. ACM, 2002.

Philip Levis, David Gay, and David Culler. Active sensor networks. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 343–356, Berkeley, CA, USA, 2005a. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251203.1251228`.

Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005b.

N. Li and J. Offutt. An Empirical Analysis of Test Oracle Strategies for Model-Based Testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 363–372, March 2014. doi: 10.1109/ICST.2014.49.

Yu-Cheng Norm Lien and Wen-Jong Wu. NTUPreter: High-Level Structured Programming Platform for Wireless Sensor Networks. *International Journal of Electronics and Electrical Engineering*, 2(2):138–142, June 2014. doi: 10.12720/ijeee.2.2.138-142.

M.K. Maggs, S.G. O'Keefe, and D.V. Thiel. Consensus Clock Synchronization for Wireless Sensor Networks. *Sensors Journal, IEEE*, 12(6):2269–2277, June 2012.

Mihai Marin-Perianu and Paul Havinga. *Ubiquitous Computing Systems: 4th International Symposium, UCS 2007, Tokyo, Japan, November 25-28, 2007. Proceedings*, chapter D-FLER – A Distributed Fuzzy Logic Engine for Rule-Based Wireless Sensor Networks, pages 86–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-76772-5. doi: 10.1007/978-3-540-76772-5_7. URL `http://dx.doi.org/10.1007/978-3-540-76772-5_7`.

Aline Freitas Martins and Ricardo de Almeida Falbo. Models for Representing Task Ontologies. In *Proceeding of the 3rd Workshops on Ontologies and their Application*, 2008.

Gloria Martorella, Daniele Peri, and Elena Toscano. Hardware and Software Platforms for Distributed Computing on Resource Constrained Devices. In Salvatore Gaglio and Giuseppe Lo Re, editors, *Advances onto the Internet of Things*, volume 260 of *Advances in Intelligent Systems and Computing*, pages 121–133. Springer International Publishing, 2014. ISBN 978-3-319-03991-6. doi: 10.1007/978-3-319-03992-3_9.

Gilles Mauris, Eric Benoit, and Laurent Foulloy. Fuzzy Symbolic Sensors-From Concept to Applications . *Measurement*, 12(4):357 – 384, 1994. ISSN 0263-2241.

Chris Miller and Christian Poellabauer. Reliable and Efficient Reprogramming in Sensor Networks. *ACM Trans. Sen. Netw.*, 7(1):6:1–6:32, August 2010. ISSN 1550-4859. doi: 10.1145/1806895.1806901. URL `http://doi.acm.org/10.1145/1806895.1806901`.

J. Scott Miller, Peter A. Dinda, and Robert P. Dick. Evaluating a BASIC Approach to Sensor Network Node Programming. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 155–168, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-519-2. doi: 10.1145/1644038.1644054.

Naouel Moha, Yann-GaÃ«l GuÃ©hÃ©neuc, Anne-FranÃ§oiseLe Meur, Laurence Duchien, and Alban Tiberghien. From a Domain Analysis to the Specification and Detection of Code and Design Smells. *Formal Aspects of Computing*, 22 (3-4):345–361, 2010. ISSN 0934-5043. doi: 10.1007/s00165-009-0115-x.

M.Trute. AmForth Documentation, 2016. Available online at `http://amforth.sourceforge.net/amforth.pdf`.

W. Munawar, M.H. Alizai, O. Landsiedel, and K. Wehrle. Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–6, May 2010. doi: 10.1109/ICC.2010.5501964.

M Navara and D Peri. Automatic Generation of Fuzzy Rules and its Applications in Medical Diagnosis. In *Proc. 10th Int. Conf. Information Processing and Management of Uncertainty, Perugia, Italy*, volume 1, pages 657–663, 2004.

Richard Oliver, Adriana Wilde, and Ed Zaluska. Reprogramming Embedded Systems at Run-time. In *Proceedings of the 8th International Conference on Sensing Technology*, pages 124–129, 2014.

Pankesh Patel and Damien Cassou. Enabling high-level application development for the internet of things. *Journal of Systems and Software*, 103:62 – 84, 2015. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2015.01.027. URL `http://www.sciencedirect.com/science/article/pii/S0164121215000187`.

Stephen Pelc. Programming Forth. *Microprocessor Engineering Limited*, 2011.

Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. The Evolution of Forth. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 177–199, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4. doi: 10.1145/154766.155369. URL `http://doi.acm.org/10.1145/154766.155369`.

Priyanka Rawat, KamalDeep Singh, Hakima Chaouchi, and JeanMarie Bonnin. Wireless Sensor Networks: a Survey on Recent Developments and Potential Synergies. *The Journal of Supercomputing*, 68(1):1–48, 2014. ISSN 0920-8542. doi: 10.1007/s11227-013-1021-9. URL `http://dx.doi.org/10.1007/s11227-013-1021-9`.

Andrew Read. Concept and Implementation of an Extended Return Stack to Enhance Subroutine and Exception Handling in FORTH. In *30th EuroForth Conference*, pages 5–22, 2014.

J.M. Serrano, J. Serrat, and J. Strassner. Ontology-Based Reasoning for Supporting Context-Aware Services on Autonomic Networks. In *Communications, 2007. ICC '07. IEEE International Conference on*, pages 2097–2102, June 2007. doi: 10.1109/ICC.2007.347.

Muzammil Shahbaz, K. C. Shashidhar, and Robert Eschbach. Iterative Refinement of Specification for Component Based Embedded Systems. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 276–286, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0562-4. doi: 10.1145/2001420.2001454.

Gabor Simko, David Lindecker, Tihamer Levendovszky, Sandeep Neema, and Janos Sztipanovits. *Specification of Cyber-Physical Components with Formal Semantics – Integration and Composition*, pages 471–487. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

Gabor Simko, Tihamer Levendovszky, Miklos Maroti, and Janos Sztipanovits. Towards a Theory for Cyber-Physical Systems Modeling. In *Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, pages 56–61. ACM, 2014.

Mark-Oliver Stehr and Carolyn L Talcott. Plan in Maude Specifying an Active Network Programming Language. *Electronic notes in theoretical computer science*, 71:240–260, 2004.

Bill Stoddart, Campbell Ritchie, and Steve Dunne. Forth Semantics for Compiler Verification. In *28th EuroForth Conference*, pages 45–58, 2012.

R. VanNorman. Fuzzy Forth. *Forth Dimensions*, 18:6–13, March 1997. ISSN 0884-0822.

V. Vyatkin. Software Engineering in Industrial Automation: State-of-the-Art Review. *IEEE Transactions on Industrial Informatics*, 9(3):1234–1249, Aug 2013. ISSN 1551-3203. doi: 10.1109/TII.2013.2258165.

Hiroshi Wada, Pruet Boonma, Junichi Suzuki, and Katsuya Oba. Modeling and Executing Adaptive Sensor Network Applications with the Matilda UML Virtual Machine. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, pages 216–225. ACTA Press, 2007.

Li Da Xu, Wu He, and Shancang Li. Internet of things in industries: A survey. *Industrial Informatics, IEEE Transactions on*, 10(4):2233–2243, Nov 2014. ISSN 1551-3203. doi: 10.1109/TII.2014.2300753.

Yeung Yam, P. Baranyi, and Chi-Tin Yang. Reduction of Fuzzy Rule Base via Singular Value Decomposition. *Fuzzy Systems, IEEE Transactions on*, 7(2): 120–132, Apr 1999. ISSN 1063-6706. doi: 10.1109/91.755394.

Tingting Yu, Ahyoung Sung, Witawas Srisa-an, and Gregg Rothermel. An approach to testing commercial embedded systems. *Journal of Systems and Software*, 88:207 – 230, 2014. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2013.10.041. URL `http://www.sciencedirect.com/science/article/pii/S0164121213002604`.

Xi Zheng and Christine Julien. Verification and Validation in Cyber Physical Systems: Research Challenges and a Way Forward. In *Proceedings of the First International Workshop on Software Engineering for Smart Cyber-Physical Systems*, SEsCPS '15, pages 15–18, Piscataway, NJ, USA, 2015. IEEE Press. URL `http://dl.acm.org/citation.cfm?id=2821404.2821410`.

Y. Zuo, G. Sun, C. Ouyang, and G. Yang. Evaluating CTP in Energy-Harvesting Wireless Sensor Networks: An Experimental Study. In *Network and Information Systems for Computers (ICNISC), 2015 International Conference on*, pages 26–33, Jan 2015. doi: 10.1109/ICNISC.2015.129.