Using Event-Based and Rule-Based Paradigms to Develop Context-Aware Reactive Applications

Truong Giang Le

CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS | le cnam

École Doctorale Informatique, Télécommunications et Électronique (Paris)

PHD THESIS

presented by: **TRUONG GIANG LE**

defense date: **September 30, 2013**

Submitted in partial fulfillment of the requirements for the degree of: **Doctor of Philosophy in Conservatoire National des Arts et Métiers**

Major/Specialization: **Computer Science**

Using Event-Based and Rule-Based Paradigms to Develop Context-Aware Reactive Applications

Thesis committee

Thesis Director	Renaud Rioboo	*Full Professor, ENSIIE, France*
Thesis Supervisor	Matthieu Manceny	*Associate Professor, ISEP, France*
Thesis Supervisor	Olivier Hermant	*Assistant Professor, MINES ParisTech, France*
Thesis Supervisor	Renaud Pawlak	*Technical Director - PhD, IDCapture, France*
Reviewer	Pascale Le Gall	*Full Professor, ECP, France*
Reviewer	Carlos E. Cuesta	*Associate Professor, URJC, Spain*
Examiner	Samia Bouzefrane	*Full Professor, CNAM Paris, France*
Examiner	Houman Younessi	*Full Professor, RPI, USA*

"The function of good software is to make the complex appear to be simple."
Grady Booch

Acknowledgement

My thesis is the result of the work which was carried out mostly at the laboratories CEDRIC (CNAM-ENSIIE) and LISITE-ISEP. It would not have been possible to complete this doctoral thesis without the encouragement, help, and support from many people around me. I would like to express my profound sense of reverence to all of them, not only to some people mentioned here.

First of all, I would like to express my sincere thanks and respect to Professor Renaud Rioboo for his expert and patient guidance during my doctoral research during past three years. It is my great honor to be mentored by him. As my thesis's director, I have learnt extensively from him, including the strong enthusiasm for doing research, idea development, proposing convincing arguments and problem-solving techniques.

My second and sincere appreciation goes to my supervisors, Professor Matthieu Manceny, Professor Olivier Hermant, and Dr. Renaud Pawlak. I owe them a lot of gratitude for their valuable comments and suggestions throughout my studies. Their constant guidance inspired and motivated me, and also made my graduate career done on the right foot. I want to convey a great thank to them.

I am especially indebted to my dissertation committee members, Professor Pascal Le Galle, Professor Samia Bouzefrane, Professor Carlos E. Cuesta, and Professor Houman Younessi, for the time they spent on proofreading of my manuscripts and evaluating my thesis. Their constructive feedbacks help me to significantly improve my work.

I am fortunate to have been surrounded by a wonderful group of excellent colleagues in my laboratories, CEDRIC and LISITE. This is my pleasure to have a chance to work and discuss with them various interesting technical topics. Special thanks go to Professor Amara Amara, Professor Raja Chiky, Professor Catherine Dubois, Sylvain Lefebre, etc. I would like to acknowledge the financial support of the European Union for my doctoral funding inside the scope of the MCUBE project.

To my dearest friends, many thanks for your helpfulness constant support. My life in Paris began since 2010. Afterward, I have enjoyed my stay with a lot of good friends. Besides, I have many other friends in Vietnam, Korea, USA, England, etc. With their shares, they made me feel happy and avoid feeling homesick.

Last, but by no means least, I take this opportunity to express the profound gratitude from my deep heart to my family for their endless love and encouragement. It is the truth that I always have my family to count on when times are tough. They are the most basic source of inspiration and motivation so that I have enough belief and strength to keep going and accomplish my work. This thesis is dedicated to them.

Abstract

Context-aware pervasive computing has attracted a significant research interest from both academy and industry worldwide. It covers a broad range of applications that support many manufacturing and daily life activities. For instance, industrial robots detect the changes of the working environment in the factory to adapt their operations to the requirements. Automotive control systems may observe other vehicles, detect obstacles, and monitor the essence level or the air quality in order to warn the drivers in case of emergency. Another example is power-aware embedded systems that need to work based on current power/energy availability since power consumption is an important issue. Those kinds of systems can also be considered as smart applications. In practice, successful implementation and deployment of context-aware systems depend on the mechanism to recognize and react to variabilities happening in the environment. In other words, we need a well-defined and efficient adaptation approach so that the systems' behavior can be dynamically customized at runtime. Moreover, concurrency should be exploited to improve the performance and responsiveness of the systems. All those requirements, along with the need for safety, dependability, and reliability pose a big challenge for developers.

In this thesis, we propose a novel programming language called INI, which supports both event-based and rule-based programming paradigms and is suitable for building concurrent and context-aware reactive applications. In our language, both events and rules can be defined explicitly, in a stand-alone way or in combination. Events in INI run in parallel (synchronously or asynchronously) in order to handle multiple tasks concurrently and may trigger the actions defined in rules. Besides, events can interact with the execution environment to adjust their behavior if necessary and respond to unpredictable changes. We apply INI in both academic and industrial case studies, namely an object tracking

program running on the humanoid robot Nao and a M2M gateway. This demonstrates the soundness of our approach as well as INI's capabilities for constructing context-aware systems. Additionally, since context-aware programs are wide applicable and more complex than regular ones, this poses a higher demand for quality assurance with those kinds of applications. Therefore, we formalize several aspects of INI, including its type system and operational semantics. Furthermore, we develop a tool called INICheck, which can convert a significant subset of INI to Promela, the input modeling language of the model checker SPIN. Hence, SPIN can be applied to verify properties or constraints that need to be satisfied by INI programs. Our tool allows the programmers to have insurance on their code and its behavior.

Keywords: event-based programming, rule-based programming, context-aware pervasive computing, smart computing, robot programming, concurrent programming, embedded programming, verification and validation, static analysis, model checking.

Résumé

Les applications réactives et sensibles au contexte sont des applications intelligentes qui observent l'environnement (ou contexte) dans lequel elles s'exécutent et qui adaptent, si nécessaire, leur comportement en cas de changements dans ce contexte, ou afin de satisfaire les besoins ou d'anticiper les intentions des utilisateurs. La recherche dans ce domaine suscite un intérêt considérable tant de la part des académiques que des industriels. Les domaines d'applications sont nombreux: robots industriels qui peuvent détecter les changements dans l'environnement de travail de l'usine pour adapter leurs opérations ; systèmes de contrôle automobiles pour observer d'autres véhicules, détecter les obstacles, ou surveiller le niveau d'essence ou de la qualité de l'air afin d'avertir les conducteurs en cas d'urgence ; systèmes embarqués monitorant la puissance énergétique disponible et modifiant la consommation en conséquence. Dans la pratique, le succès de la mise en œuvre et du déploiement de systèmes sensibles au contexte dépend principalement du mécanisme de reconnaissance et de réaction aux variations de l'environnement. En d'autres termes, il est nécessaire d'avoir une approche adaptative bien définie et efficace de sorte que le comportement des systèmes peut être modifié dynamiquement à l'exécution. En outre, la concurrence devrait être exploitée pour améliorer les performances et la réactivité des systèmes. Tous ces exigences, ainsi que les besoins en sécurité et fiabilité constituent un grand défi pour les développeurs.

C'est pour permettre une écriture plus intuitive et directe d'applications réactives et sensibles au contexte que nous avons développé dans cette thèse un nouveau langage appelé INI. Pour observer les changements dans le contexte et y réagir, INI s'appuie sur deux paradigmes : la programmation événementielle et la programmation à base de règles. Événements et règles peuvent être définis en INI de manière indépendante ou en

combinaison. En outre, les événements peuvent être reconfigurés dynamiquement au cours de l'exécution. Un autre avantage d'INI est qu'il supporte la concurrence afin de gérer plusieurs tâches en parallèle et ainsi améliorer les performances et la réactivité des programmes. Nous avons utilisé INI dans deux études de cas : une passerelle M2M multimédia et un programme de suivi d'objet pour le robot humanoïde Nao. Enfin, afin d'augmenter la fiabilité des programmes écrits en INI, un système de typage fort a été développé, et la sémantique opérationnelle d'INI a été entièrement définie. Nous avons en outre développé un outil appelé INICheck qui permet de convertir automatiquement un sous-ensemble d'INI vers Promela pour permettre un analyse par model checking à l'aide de l'interpréteur SPIN.

Mots-clefs: programmation événementielle, programmation à base de règles, applications sensibles au contexte, smart computing, programmation de robots, programmation concurrente, programmation embarquée, vérification et validation, analyse statique, model checking.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Research Context

We are living in the era of pervasive computing and the role of software in our full-of-competition society has become more and more important. Software is often the key component to help companies and organizations decrease operational costs and increase profit [LB07], and has become one of the most widely used products in human history [JBS11]. Without software, professional people cannot succeed if they want to adapt and thrive in today's ever-changing global marketplace.

In recent years, in order to satisfy higher demands from customers, software systems have been required to become more robust, flexible, dependable, customizable, and self-optimizing. With more complicated and sophisticated requirements, software size and complexity have increased in a breathtaking manner. For example, the average embedded device now has one million lines of code (LOC), and that number is doubling every two years. A modern passenger jet, such as a Boeing 777, depends on 4 million LOC. Older planes such as a Boeing 747 had only 400,000 LOC [CC08]. Moreover, developing, maintaining, and ensuring quality of software systems is more difficult and resource-consuming because they are operating in an environment that is not well-defined or predictable. As a result, software errors happen frequently and cause severe damages. For example, due to a malfunction in the control software, the rocket Ariane 5's first test flight on June 4, 1996 failed [Gar05]. Annually, quality problems cost software companies up to $312 billion [Bus12]. Moreover, according to [Boe07], the cost to fix an application defect after

deployment is more than 150 times the cost to fix it in the development phase. Indeed, the longer a bug stays in the development process, the more time and effort it requires to fix. Cumulative defect removal efficiency (combinations of all defect prevention, pretest removal, testing, post-release removal) in the United States is only about 85%, so all software applications are delivered with latent defects [JBS11]. Consequently, software systems need a new and innovative approach for evolving and running [YW03].

To answer those needs, the area of formal methods, which groups together very heterogeneous rigorous approaches to systems and software development [Alm11], has been introduced. This includes a wide range of mathematically-based subjects such as formal languages, logic, knowledge representation, program semantics, type systems, formal specification, formal development, and formal verification and validation. These techniques may be applied in a systematic way at different points through the development process: requirements specification, model and design, implementation, testing and verifying [Gab06]. Using formal methods gives us some proofs on the quality of software programs. For example, when the semantics of a programming language is defined, programmers understand deeply about the behavior of their code and know exactly "what does the program do". As a result, in recent years, researchers try to propose and also extend operational semantics for classical languages like Java, C++, Ada or JavaScript.

From the user's point of view, quality is fitness for purpose or meeting user's needs and high quality means none or few problems of limited impact on customers [Tia05]. Chemuturi *et al.* [Che10] expresses the term "quality" in more details: defect-free functioning, reliability, ease of use, acceptable levels of fault tolerance during use, and safety from injury to people or property. Quality has four dimensions: specification, design and analysis, development, and conformance [Che10]. Some outstanding benefits of high software quality are (adapted from [Che10]):

- It shortens development schedules.
- It lowers development costs.
- It lowers maintenance costs.
- It reduces warranty costs.

- It increases customer satisfaction.

To develop high-quality software, programmers need robust and versatile programming languages. The first high-level ones were developed in the 1950s and since then, research in programming language has been a very attracting and active area of study in computer science. Current trends on this field are trying to find new paradigms meeting the demands of recent advances in computing and communications developments, such as multicore CPUs and pervasive computing, and help programmers to deal efficiently with the uncertainty of the environment. The compilers also need to be well-designed and optimized to obtain better dependability and performance [GvRB+12]. Moreover, industry leaders complain that modern software development yields programs that are too big, complex, and difficult to understand, particularly as customers demand more system functionality and reliability [Ort12]. As a result, new programming languages should fulfill the requirements in terms of code simplicity, quality, and maintainability. "The drive to reduce complexity is at the heart of software development" [McC04] since software complexity causes the development and maintenance costs increase significantly.

To solve the above mentioned difficulties, one popular approach is the use of Domain Specific Languages (DSLs). DSLs are languages dedicated to specific application domains or problems [Gho10, Par09]. In fact, whenever new problems are encountered, it is essential to demand new programming languages, or at least some conceptual ideas [Jon10]. By using notations and constructs designed for a particular domain, we can gain much more power and expressiveness when compared to General Purpose Languages (GPLs) [MHS05]. In other words, DSLs are very good at taking certain narrow parts of programming and making them easier to understand, and therefore quicker to write, quicker to modify, and less likely to breed bugs [Fow10].

My research is involved in building and ensuring quality for context-aware reactive applications, which are widespread with many possible useful applications for both daily life and manufacturing activities. As we will see later in Chapter 2, there is no well-defined language constructed so far to support programmers to build those kinds of applications straightforwardly and intuitively. Therefore, the object of my Ph.D. dissertation is developing a new DSL dedicated to context-aware computing. The language should come with

a strong type system and a well-defined operational semantics. Furthermore, it will be a benefit if programmers may use other formal analysis techniques during the development phase to verify and validate wanted properties of written programs.

1.2 MCUBE Project

My work is involved in the MCUBE (Multimedia 4 Machine 2 Machine) project [FED12], which is funded by a FEDER grant from the French program "Compétitivité régionale et emploi 2007-2013", co-funded by European Structural Funds. Principally, this project aims to provide a generic Machine-to-Machine (M2M) system (framework and programming language) for multimedia applications, e.g. involving sound and image collection and analysis (see Figure 1.1).

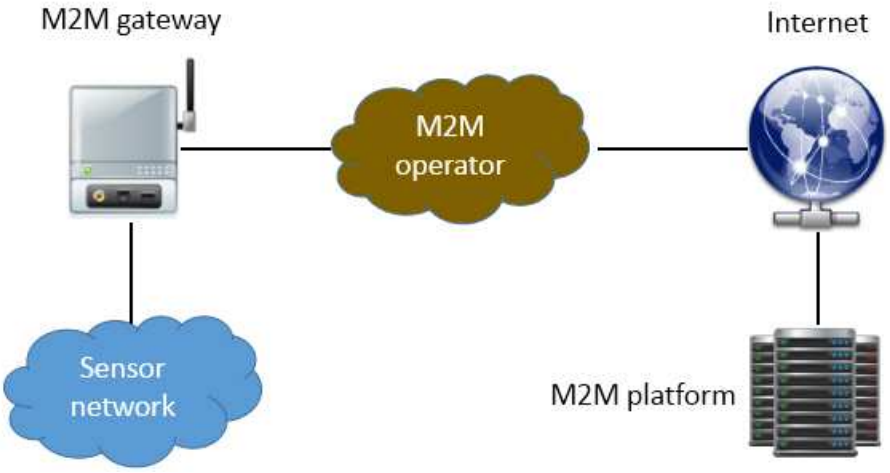


Figure 1.1: Global architecture for the MCUBE project.

1.2.1 Introduction to M2M Technologies

1.2.1.1 Overview

M2M refers to technologies that allow data communication and interaction between machine(s), device(s) or sensor(s) over a network without human intervention. The M2M connectivity market, a.k.a. the "Internet of Things", is growing worldwide. With the reduced cost for accessing wired/wireless networks, many M2M solutions and applications

have been developed in many sectors. Figure 1.2 shows the three waves of connected devices, in which the current trend is making everything connected together. Analysts predict that there will be 25 billion connected IP devices by 2015, with M2M traffic expected to grow by 258% [Net11a]. Another survey/study estimates that M2M will generate $35 billion in service revenues by 2016 [Res11b].

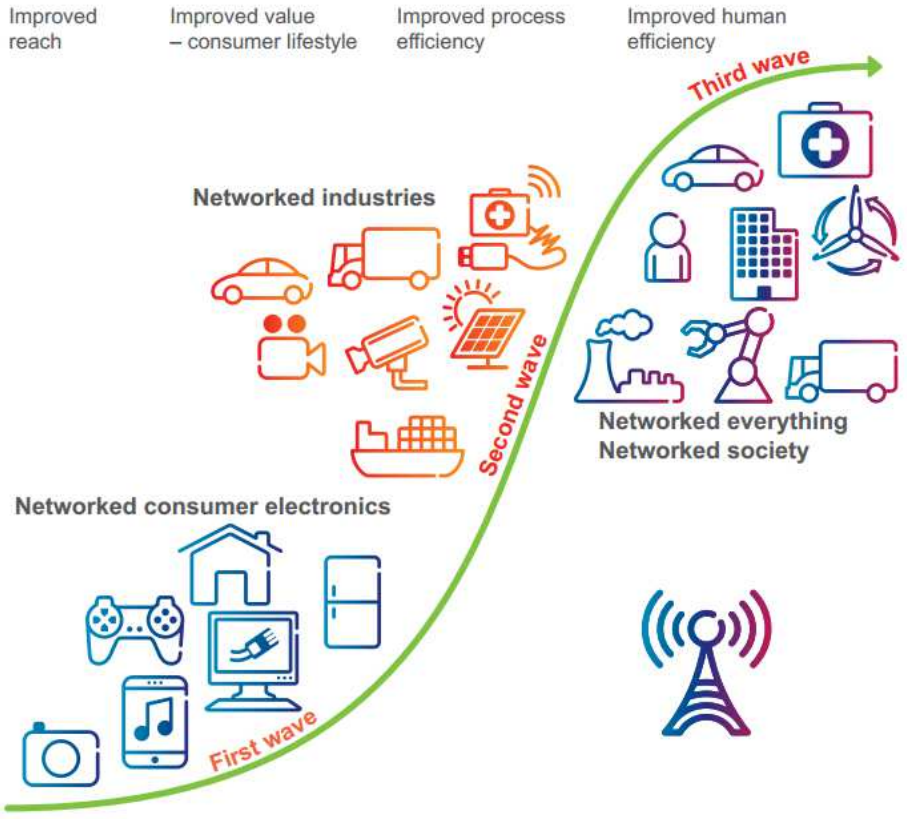


Figure 1.2: The three waves of connected device development [Eri11].

Specifically, M2M devices use sensors to capture and collect useful data (e.g. image, sound, temperature, air quality, energy consumption, etc.) by schedule, and then transmit them through the network to other devices. Applying M2M solutions brings several advantages:

- All operations can be done automatically without human supervision and effort.
- Decrease administrative and operational expenses, especially in rural areas thanks to wireless network.

Figure 1.3: M2M applications domains.

Possible M2M applications domains are illustrated in Figure 1.3 and detailed as follows [Res11a, FT12, KAK+10]:

- Construction and building: Heating, Ventilation, and Air Conditioning (HVAC) systems, security surveillance, lighting, fire, and safety systems, etc.
- Consumer and home: digital set-top boxes, hand-held devices, computers, home residential utility meters, etc.
- Industry: assets monitoring and management, fabrication, packaging, etc.
- Healthcare and life science: telemedecine, healthcare monitoring, freezers, etc.
- Energy: smart energy monitoring and controlling.
- Retail: logistics, vending machines, service equipment (e.g. gas pump, refrigeration, etc.), screen displays, etc.

- Transportation: telematics and mobile communications with vehicles (e.g. trucks, cars, trains, aircraft, ships, etc.).
- Security/public safety: emergency services, tracking and surveillance (e.g. Closed-circuit Television (CCTV) cameras).
- IT and network: network monitoring, server monitoring, resource transferring, multimedia communication, internet-based services, etc.

1.2.1.2 Research on M2M Technologies

Many M2M infrastructures frameworks, models, paradigms and services have been proposed so far to ease the development of M2M systems. For example, Herstad *et al.* [HNS+09] defined a service platform architecture for connected objects. The architecture exhibits a number of features to support scalability, rapid development, and technology and device independence. Cristaldi *et al.* [CFGO05] presented an interface platform, which is able to collect and process data from a wide variety of sensors and exchange information supporting different communication networks and protocols. Matson *et al.* [MM11] tried to create a model and architecture to support networking, communication, interaction, organization and collective intelligence features between machines, robots, software agents, and humans.

Currently, in order to build M2M applications, developers use classical programming languages (e.g. Java, .Net, C/C++, Perl, etc.) or extensions of them. Besides, there have been several works on constructing event-based programming languages [CK08, HZJE11], which also can be applied to handle events happening in M2M systems (see Section 2.2, page 39). However, these languages are not fully comfortable for M2M applications since they lack a well-defined mechanism to support scheduled operations, which are essential in M2M communication. Another limitation is that events are not constructed and handled in an intuitive manner, i.e. they are mixed with other syntaxes and notations.

Although M2M has attracted a large amount of attention over the years, developing M2M applications is still challenging. Besides, "existing M2M solutions are fragmented and usually are dedicated to a specific single application" [Net11b]. Currently, M2M industry

continues to looks for better and more comprehensive M2M solutions.

To understand more about M2M technologies, especially with challenges, state-of-the-art achievements and future trends, interested readers may refer to [HBE11, Roe11, BEH12].

1.2.2 Goal of the MCUBE Project

The goal of our project is to use standard video and sound capture devices for supervision applications in domains such as agriculture and photovoltaic energy. For example, image analysis can be used to follow crop growth in a field, or to check the state of solar panels for maintenance. Sound analysis may be used to count insects in a field, to detect intrusions in plants, etc. The project MCUBE will provide a service for finding, deploying and verifying such algorithms on M2M applications such as M2M gateways, which allow the simultaneous capture and transfer of multimedia and various sensor data. In order to operate more efficiently, the gateway should capture, handle and process data in a well-defined and robust way.

In the MCUBE project, my thesis consists of studying the use of advanced and formal modeling and implementation techniques in order to ensure the quality of M2M applications. These applications often need to take into account heterogeneous asynchronous data flows coming from sensor networks to which they are connected. In order to ensure the quality and the consistent deployment and administration of such applications, it is necessary to ensure a certain degree of formal or semi-formal validation with regard to their environment. The goal of this thesis will be to invent methods and tools to achieve this validation (including for example DSLs, appropriated environment models, formal validation techniques, etc.). The approach can be language-based or model-driven, or both. An important challenge of this thesis is to invent a technique that can be used by industrial firms at a relatively low cost. For example, platforms that are free to use and develop, static analysis techniques and partial interpretation are interesting candidates. Nevertheless, our objective does not only target at M2M applications, but also context-aware applications in general.

1.3 Motivation, Purpose and Proposed Approach

Although several programming languages have been proposed to support the building of context-aware reactive applications, their supports and capabilities are still limited. For instance, the context-change recognition is not defined in an intuitive way and changes cannot be handled in parallel efficiently. Moreover, written programs cannot dynamically modify their behavior at runtime to adapt to new requirements and constraints.

Our goal is to develop a new programming language, which may aid programmers to write conveniently and robustly those kinds of systems that need to take advantage of context-awareness and multithreading such as M2M systems, monitoring and controlling systems, robotic systems, autonomous systems, interactive systems, smart embedded systems/devices, manufacturing systems, etc. The developed language should have clear and well-defined syntax and semantics. Furthermore, it also should be equipped with a strong type system through a rigorous type checking mechanism. Ideally, this language also must be automatically converted to a modeling language so that users may apply model checking techniques, which become more and more popular in both academic and industry, to verify and validate desired properties.

We develop a new programming language called INI that combines both event-based and rule-based styles. These styles have been proven to be appropriate to write many kinds of applications, especially context-aware adaptive and reactive systems. Nevertheless, there are several limitations with existing languages such as the lack of well-defined context capturing and handling mechanisms or support for concurrency and dynamic adaptation when programs are running.

In our language, programmers may define events and rules independently or in combination. Events in INI run in parallel either asynchronously or synchronously. Each event has input parameters to tune the execution and output parameters that can be understood as return results. In order to facilitate the developments, we provide several built-in events. Moreover, we specify an open template so that programmers can write their own events in other languages such as Java and C/C++, and then integrate them to INI programs. Last but not least, events may be stopped or reconfigured (i.e. changing their behavior) at

runtime.

INI also comes with a flexible but strong type system so that any type conflict will be prohibited before running. We apply INI in two case studies: a M2M gateway tested on a real embedded device and an object tracking program running on the humanoid robot Nao, which demonstrate INI's capabilities. Furthermore, in order to help programmers to verify and validate their programs more straightforwardly, we build a tool to convert a significant subset of our language to Promela, the modeling language of the model checker SPIN. Then SPIN can be used to check important constraints and properties that need to be satisfied. This makes INI programs more reliable.

1.4 Organization of This Dissertation

The content of this thesis is organized as follows. In Chapter 2, we introduce the state-of-the-art in the field of research on context-aware pervasive computing, along with event-based and rule-based programming styles. Next, in Chapter 3, an overview of INI with major features and several examples is presented. Later, we give a formalization approach to define an operational semantics of INI in Chapter 4. The type system of INI and our supported tool INICheck for model checking INI programs are discussed in Chapter 5. Then two case studies of applying INI are shown in Chapter 6. Chapter 7 concludes my thesis.

Chapter 2

Background

The purpose of this chapter is to provide a background knowledge on context-aware pervasive computing (Section 2.1), event-based programming (Section 2.2), and rule-based programming (Section 2.3). For those topics, we discuss recent trends, ideas and supports, along with their advantages and disadvantages.

2.1 Context-Aware Pervasive Computing

2.1.1 Overview

Context and Context-Awareness The notion of context has been observed in numerous areas that cover all operations and activities of humans and systems. The Cambridge Advanced Learner Dictionary defines context as "the situation within which something exists or happens, and that can help explain it" [Cam08]. According to the Concise Oxford English Dictionary, context is "the circumstances that form the setting for an event, statement, or idea, and in terms of which it can be fully understood" [SS06]. Over the past years, many researches in computer science also provided a vast and diverse number of definitions. For example, context is regarded as "any information that can be used to characterize the situation of an entity", in which an entity is "a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves" [DAS01]. In other words, context can include information about inside states of the system or many different kinds of sensed data related to the interaction between humans, applications and the surrounding environment: location,

identity of user, activity, time, motion, light level, sound level, observed phenomena, etc [Cha11, DAS01]. These information are not known in advance when our program is written. As a result, context-awareness is really useful and necessary in practice. According to [Tuu00], there exist two categories of context-awareness:

- Self-contained context-awareness: no need of outside supports to achieve context-awareness (i.e. there is a complete internal mechanism for recognizing and handling contexts).
- Infrastructure-based context-awareness: outside supports (e.g. from other autonomous systems) are needed to achieve context-awareness.

Actually, context-awareness is a central feature of ubiquitous systems [Kru09, Sat09], and is considered as one type of intelligent computing [KD06, Cur11].

Context-Aware Pervasive Computing Context-aware computing was first discussed by Schilit *et al.* [SAW94]. Essentially, it is a study of building intelligent applications that can monitor the running context by registering event handlers. Dey [Dey00, Dey01] categorized two types of context-aware computing:

- Using context: just exploiting context information.
- Adapting to context: this is a higher level since our systems not only use the context but also adapt to it. In case of changes in the context (i.e. some events occur), systems may (automatically) adapt their behavior if needed and react accordingly in order to satisfy the user's current needs or anticipate the user's intentions [HP85, DSFv09].

The ultimate goal of context-aware computing is reducing the burden of excessive user involvement and providing proactive intelligent assistance [Lok06]. Context-aware applications look at what activities are occurring with entities and use this information to determine why a situation is happening [Kru09]. Such aware systems have become one of the most exciting concepts in ubiquitous (pervasive) computing, with a wide range of application areas such as autonomous systems, monitoring and controlling programs,

robots, mobile applications, location-based services, health care systems, manufacturing systems, interactive programs, etc [Lok06, Dar09, MdRM09]. For instance, in medical centers, patient data such as blood pressure, heart rate, body temperature, and respiratory rate should be accumulated and monitored periodically so that when abnormal symptoms occur, the doctors are notified. In a smart home, the hot water level, quality of air, humidity level or temperature can be monitored and controlled to provide good living conditions to the owner. Another example is smart robots, which must recognize changes in the environment or user's activities in order to adopt a suitable behavior. In those cases, taking advantage of context-awareness helps us improve the "cleverness" and responsiveness of our programs. Context-awareness is a primitive concept that advocates for a programming language supporting those features natively.

Since then, context-aware systems have received a worldwide attention from both academy and industry [SGP12]. There have been numerous attempts to support and upgrade context-aware computing in order to satisfy requests from customers. However, developing context-aware reactive applications is still difficult and challenging [ST09]. The main reason is that we need a well-defined mechanism to handle efficiently widely varied sources of context information [CBC+04].

Besides, context-aware systems should be able to handle multiple work in parallel due to the inherent asynchronous and overlapping nature of the environment context. With context-aware systems, several events can occur simultaneously, and thus this requires parallelism for the system to react in a sound and efficient way. In computer science, concurrent programming allows one program to execute multiple tasks simultaneously, by dividing it into multiple threads and taking advantage of parallelism. This kind of processing has been shown to be a powerful approach to speed up the execution, solve larger problems, and improve the performance and responsiveness of software systems. With the spread of multicore and multiprocessor technologies, concurrent programming is going to be mainstream as more programmers are confronted with threading [San11, Bre09, HS08]. However, creating multithreaded applications is hard since they may contain bugs that are notoriously difficult to find and fix [CT05, SBS+11]. As a result, equipping context-ware systems with multithreading is a non-trivial task.

To understand more about context-aware pervasive computing, interested readers may refer to [Kru09, Lok06, Dar09].

2.1.2 Categories of Context Information

In general, a context includes all information and data that may be collected when our program is running. Several authors tried to classify those information so that managing and handling context will be easier.

For instance, Chen *et al.* [CK00] distinguished four context categories:

- Computing context: system state, network connectivity, bandwidth, power level, etc.
- User context: user's profiles, user's location, user's mobility, user's behavior, etc.
- Physical context: lighting, temperature, humidity, etc.
- Time context: day, week, year, season, etc.

Similarly, Raz *et al.* [RJSF06] divided context information into four categories:

- User context: user information and profiles.
- Location context: the information related to the location of the user.
- Application context: this context is related to the applications that the end user is using.
- Network context: this is the relevant information about the networks that are available in the user's location.

In brief, we can see that there are two main kinds of context:

- **Internal context**: all information related to the data and state inside the system itself such as the power level, the system status (e.g. stable or unstable), resource consuming, variables' values, memory state, etc.
- **External context**: all information related to the surrounding environment for the system like the user context, the physical context, the time context, etc.

Ideally, the context-aware mechanism should be able to recognize and capture as much context information as possible so that the system may deal efficiently and flexibly with various situations. Our approach will consider all those kinds of context information as shown later.

2.1.3 Categories of Context-Aware Adaptation

Context-aware adaptation is naturally an event-driven behavior, which means that when a change in the context is detected, actions or reconfigurations of the system are triggered [Cha11]. Adaptation can also be mapped to evolution [ST09]. Buckley *et al.* [BMZ+05] provided a taxonomy of evolution based on the object of change (`where`), system properties (`what`), temporal properties (`when`), and change support (`how`).

In order to explain what systems may adapt to, Miraoui *et al.* [MTFA11] mentioned four main categories of adaptation:

- Content adaptation: for instance, a smart TV may show favorite programs/channels for different users.

- Behavior adaptation: for instance, a system may change its behavior to save energy when low-energy situation is encountered.

- Interface adaptation: for instance, a website may change its theme in different countries.

- Software component adaptation: for instance, one software component changes its settings to work well with another one.

2.1.4 Solutions for Context-Aware Adaptation

Several kinds of adaptation solutions have been defined so far. In the following parts, we will detail some recent approaches.

2.1.4.1 Architecture-Based Adaptation

Choi *et al.* [Cho08] presented a software architecture, named WCAM [1], which lessens the complexity of developing context-aware systems by decoupling concerns. The Watcher perceives the outside environment, and the Controller manages the collaboration between the Watcher and the Model. The Model collects contextual information, interprets the information, and then generates the system contexts. The role of the Action is managing services that are related to contexts. Lopes *et al.* [LF05] proposed a formal approach for designing context-aware systems. This approach is based on the establishment of a set of primitives through which context notion can be modeled as a first-class entity. Furthermore, context-awareness is addressed explicitly as an additional dimension of architectural elements. Hussein *et al.* [HHCY12] introduced a layered architecture that supports developing context-aware and self-adaptive systems. Their approach aims not only to adapt to the system but also to model, process, and manage the context information. This layered architecture, the system and its context components, the system and its context representation and the change management, divide the system in an appropriate manner so that all the context-aware system's requirements can be dealt with different layers. Vales-Alonso *et al.* [VAELMnG+08] presented UCare [2], a context-aware and flexible architecture that provides disabled users with abundant environment adaptation services. The authors have considered bidirectional adaptability, which means adapting the system interface to users' capabilities and also the system's features to users' needs.

A more comprehensive literature review for architecture-based adaptation can be found in [CdLG+09].

2.1.4.2 Model-Driven Adaptation

Chen *et al.* [CTP+10] described a model-based engineering approach to support the development of self-configuring embedded systems. In their work, some popular operations such as upgrades, attachment of devices, relocation of applications and adjustment of performance parameters can be carried out during runtime for various kinds of purposes

[1] WCAM stands for "Watcher, Controller, Action, and Model".
[2] UCare stands for "Urban Care".

such as information/function integration, maintenance, performance, resource efficiency, and robustness. Hussein *et al.* [HHC11] introduced a methodology to model and realize context-aware adaptive software systems. The proposed approach explicitly separates the context model and the system model. However, their relationships, changes, and change impacts across the system and its contexts can be conveniently captured and managed. Magableh *et al.* [MB09] presented a model to dynamically compose adaptable context-dependent applications using context conditions. This component-based approach allows the modification of the application architecture by subdividing components into subsystems of static and dynamic elements.

Interested readers may find a more comprehensive literature review for model-driven adaptation in [NDR09].

2.1.4.3 Automated Learning-Based Adaptation

Cioara *et al.* [CAS+10] addressed the context adaptation problem by constructing a self-healing model. In this model, they applied a policy-driven reinforcement learning mechanism to take runtime decisions. The self-healing property is enforced by monitoring the system's execution environment for evaluating the degree of adequacy to the context policies in the current situation. Then the healing actions can be selected for execution. Tsang *et al.* [TC07] devised a personalized, dynamic, and runtime approach to adaptation. Their approach provides several techniques for selecting the relevant information based on the users' behavior history. Thereafter, this information may be used for mining usage patterns, and also for generating, prioritizing, and selecting adaptation behavior. Mohomed *et al.* [MCCDL06] showed how to exploit user interaction to learn how to adapt contents based on the contexts. They introduced FCS [3], an automatic technique that leverages user interactions to specify the context that has the most impact on adaptation requirements. Besides, context-awareness was added in order to make adaptation predictions for a user thanks to the history of the group of users that share the context identified by FCS.

A more comprehensive literature review for automated learning-based adaptation can be found in [Chi10].

[3] FCS stands for "Feedback-driven Context Selection".

2.1.4.4 Policy-Based Adaptation

Cioara *et al.* [CASD09] proposed a generic policy-based self-management model which can be used to automatically detect and repair problems happening during the context adaptation process. In their work, they defined a generic context policy representation model and its associated reasoning language conversion model for runtime evaluation in order to efficiently capture and evaluate the dynamic rules that govern the adaptation processes. Ouyang *et al.* [OxSD+09] introduced a policy-based framework for self-adaptive schemes in pervasive computing. To support the design of policy (the general idea is based on the Separation of Concerns (SoC) principle), the authors constructed their own expressive and extensible policy ontology and policy language. Zhu *et al.* [ZKSL09] introduced the design, implementation and evaluation of an efficient policy system called Finger. Finger allows policy interpretation and enforcement on distributed sensors to support sensor level adaptation and fine-grained access control. Moreover, dynamic management of policies, minimization of resources usage, high responsiveness and node autonomy are also featured. The authors tried to integrate their system as a component of TinyOS, a sensor operating system [All12].

2.1.4.5 Static Adaptation vs. Dynamic Adaptation

Generally, static adaptation relates to the redesign, reconfiguration of application architectures and components, and redeployment when functional and non-functional requirements from customers change, while dynamic adaptation happens at runtime due to changing resources (e.g. memory, energy, network bandwidth, external devices, etc.) and context conditions [GBE+09]. For instance, moving a system from a traditional desktop environment to a cloud or refactoring legacy code is viewed as static adaptation. One example for dynamic adaptation is that a laptop or a mobile automatically reduces the brightness when the power level is low to save energy. Since there are many changes which may happen when our program is running and that they cannot be well-predicted at the design time, dynamic adaptation has attracted more attention from researchers [RC02, BCC+07, RS08, FC09]. To understand more about static/dynamic adaptation, readers may refer to [CMP04, CMP06, ST09, LK11].

2.1.5 Programming Language Support for Context-Aware Adaptation

General purpose languages like Java, C/C++, .Net or Python can be used for programming context-aware reactive applications [Bar05, DW08, DL05, Rak03, Nat12, PBHS03, CCP08]. Since the need of developing context-aware applications is increasing, several authors try to extend classical languages so that they support building context-aware programs more easily such as ContextJ for Java [AHHM11], AwareC# for C# [RS06], ContextR for Ruby [Sch08], ContextPy for Python [HPSA10], etc. Although those languages or extensions can be applied for this purpose, they are still not convenient for developers since new notations or concepts are mixed with old ones. In other words, the methodology is not so intuitive and straightforward. Consequently, writing and maintaining context-aware applications still take a lot of time and efforts.

Besides, using event-based and rule-based programming languages is also suitable for developing context-aware adaptive and reactive systems [AHM+10, KT10, Kru09]. Therefore, in the next sections, we will present state-of-the-art in the field of event-based and rule-based programming paradigms with several notable candidates. To learn more about programming language support for context-aware adaptation, interested readers may refer to [AHH+09, SGP12].

2.2 Event-Based Programming

2.2.1 Overview

Event-based programming is a programming style where the flow of execution is determined by events. Events are handled by handlers or callbacks. An event callback is a function that is invoked when an event (i.e. something significant) happens [DZK+02, Tei12].

Events are typically used to monitor changes happening in the environment or for time scheduling. In other words, any form of monitoring can be considered to be compatible with event-based style [MFP06]. Generally, three types of events are distinguished [MFP06]:

- Timing events to express the passing of time.
- Arbitrary detectable state changes in a system, e.g. the value of a variable changes

during execution.

- Physical events such as the appearance of a person detected by cameras.

For example, programmers may define an event to monitor the power level of their system or to observe users' behavior in order to react. Other examples are using events for: stock monitoring and analysis, location monitoring, weather forecasting services, health care monitoring, security surveillance, server monitoring, network intrusion detection, etc. Programmers can also specify an event to schedule a desired action at a specific time. For instance, a program needs to send a report to the users on every Tuesday.

In recent years, event-driven programming has become pervasive as an efficient method for interacting and collaborating with the environment in ubiquitous computing. Using event-driven style requires less effort [BD04] and may lead to more robust software [DZK+02]. This style is strong and convenient to write many kinds of applications: M2M applications, sensor applications, mobile applications, simulation systems, embedded systems, robotics, context-aware reactive applications, self-adaptive systems, autonomous systems, etc. As a result, software engineering research has been motivated to pay more attention to this programming paradigm. To understand more about event-based style, especially its advantages and applications, interested readers may refer to [MFP06, Fai06, Obe05, EN10, SHX11, HB10, Fer06]. In the following sections, we introduce several event-based programming languages/extensions that were developed recently. Due to the lack of space, we choose four of them as a representative set: EventJava and TaskJava are extensions of Java to support event-driven style while EventScript and UrbiScript are stand-alone event-driven languages.

2.2.2 Event-Based Programming Languages

2.2.2.1 EventJava

Overview Eugster *et al.* [EJ09, JE09] introduced EventJava mainly as an extension of Java for event correlation. They saw that in some cases, a single and simple event can be correlated with other events, resulting in complex events. Some examples are listed below (adapted from [EJ09]):

- Average of temperature readings from some sensors inside a boiler.
- Average of temperature readings from a sensor within a five minute interval.
- The price of a stock decreases for six successive stock quotes immediately after a negative analyst report.
- Release of a new TV followed by seven positive reviews in two months.

Event correlation is essential in distributed event-based systems in which software components communicate by transmitting and receiving event notifications. By using EventJava, programmers can customize the way that events are ordered, propagated and correlated with other events.

In EventJava, an application event type is implicitly defined by declaring an event method, which can be regarded as a special kind of asynchronous method. There are two types of event attributes:

- **Explicit attributes**: These attributes represent application-specific data (e.g. value of a sensor, value of a stock quote, etc.), which can be considered as arguments in regular methods.
- **Implicit attributes**: These attributes refer to contextual information (e.g. physical or logical time, geographical or logical coordinates).

An example to show features of EventJava is shown in Listing 2.2.1 (adapted from [EJ09]). Assume that a travel agency wants to notify its customers of severe weather conditions in cities that are part of their flight itineraries. The `severeWeather` event is defined inside the `Alerts` class. This event has three attributes: `city`, `description` and `source`. The keyword `when` is used to express a *predicate* that needs to be satisfied before the *reaction* defined in the method body can be called. In other words, only those events (event method invocations) which match the *predicate* are consumed by the *reaction*. In this example, the travel agency only trusts alerts originating from the website *weather.com*. When this constraint is fulfilled, all instances of the `Alerts` class react to the simple `severeWeather` event. In this case, flight passengers' email addresses will be retrieved from a database and then, an email is sent to them.

Listing 2.2.1 *An EventJava example in which a travel agency notifies its customers about a severe weather condition.*

```
1 class Alerts {
2     ItineraryDatabase db;
3     event severeWeather(String city, String description,
4      String source)
5         when (source == "weather.com") {
6             Iterator<Itinerary> it = db.
7              getItinerariesByCity(city).iterator();
8         while(it.hasNext()} {
9             Messenger.sendEmail(it.getAssociatedEmail());
10        }
11    }
12 }
```

A `severeWeather` event can be notified to the instances of the class `Alerts` by applying one of the two following notification mechanisms, which correspond to the dynamic/static distinction:

- **Unicast**. When an event method is invoked on an object, the event is notified to that object. For example, a `severeWeather` event can be notified to an instance `a` of `Alerts` as follows:

```
a.severeWeather("Paris", "Mainly cloudy with a few
 showers", "weather.com");
```

- **Broadcast**. The same `severeWeather` event can be notified to all instances of `Alerts` as shown below:

```
Alerts.severeWeather("Paris", "Mainly cloudy with a few
 showers", "weather.com");
```

Complex Events and Correlation Patterns Besides simple events, EventJava allows programmers to define complex events by using *correlation patterns*, which are comma-separated lists of event method headers, e.g. $e_1(), e_2(), ..., e_n()$. For example, let us consider

a trading algorithm comparing `earningsReport` and `analystDowngrade` events (adapted from [EJ09]). In case that a stock has a negative earning report(i.e. the actual earnings per share, `epsAct`, is less than the estimated `epsEst`) and then an analyst downgrades to "Hold", the algorithm recommends to sell the stock (see Listing 2.2.2).

Listing 2.2.2 *Stock monitoring with EventJava.*

```
class StockMonitor {
    Portfolio p;
    event earningsReport(String firm, float epsEst,
     float epsAct, String period),
     analystDowngrade(String firm1, String analyst,
     String from, String to)
    when (earningsReport < analystDowngrade && firm == firm1 &&
     epsAct < epsEst && to == "Hold") {
        p.RecommendSell(firm);
    }
}
```

In the example above, four predicates need to be fulfilled. The first constraint `earningsReport < analystDowngrade` is used to indicate that an event `earningsReport` should happen before an `analystDowngrade` event. It is a shorthand notation for `earningsReport.time < analystDowngrade.time`. In EventJava, each event has a special *time* attribute, which is a default implicit event attribute representing time stamps for events. Implicit event attributes are fields defined globally by the `Context` class, of which an instance is passed along with every event (a simple EventJava context is shown in Listing 2.2.3). The other three constraints express the conditions related to a trading situation as we explained earlier.

Listing 2.2.3 *A simple EventJava context.*

```
public class Context implements
 Comparable<Context>, Serializable {
    public long time;
    //More fields
    public Context() { time = System.currentTimeMillis(); }
    public Context(long time) { this.time = time; }
    public int compare(Context other) {
    if (timestamp == other.timestamp) return 0;
        ...
    }
    ...
```

```
}
```

Two other examples illustrating EventJava (adapted from [JE09]) are shown below: a bank account monitor, with correlation patterns monitoring for suspicious overseas transactions (Listing 2.2.4) and a farewatcher (Listing 2.2.5).

Listing 2.2.4 *Bank account monitoring with EventJava.*

```
class AccountMonitor implements ... {
    private long debitCardNumber;
    private long account;
    private String name;
    private long SSN;
    event debitCardInactive(int numdays),
     overseasCardTransaction(float amount)
    when (debitCardInactive < overseasCardTransaction &&
     amount > 100 && numdays > 60 ) {
        alertCustomer(cardnumber);
    }
    event overseasMoneyReceipt(long id, long accountNum,
     float amount, String country),
     overseasMoneyTransfer(long id1, long accountNum1,
     float amount1, String country1)
    when(amount == amount1 && amount > 10000 &&
     (overseasMoneyTransfer.time - overseasMoneyReceipt.time)
     <= 60*60 *1000) {
        reportToIRS(amount, country, name, SSN, "Incoming");
        //Transfers > 10,000 are reported to the IRS
        reportToIRS(amount1, country1, name, SSN, "Outgoing");
        reportMoneyRouting(account, name, SSN);
    }
    ...
}
```

Listing 2.2.5 *A farewatcher written in EventJava.*

```
class FareWatcher implements ... {
    private String Address;
    event airFareDrop(String airline, String src, String dest,
     float fare),
     hotelRateDrop(String hotel, String city, String address,
     float rate)
    when (dest == city && dest == "Miami" && src == "Chicago"
     && fare <= 250 && rate <= 100) {
        sendEmail(Address, new Deal(dest, float fare,
         float rate));
    }
```

```
12      event weekendWeatherForecast(String city, String summary,
13       String detailed),
14       lastMinuteAllInclusiveDeal(String city1, float price)
15      when (city == city1 && city == "Miami" && price <= 700 &&
16       summary == "warm") {
17          sendEmail(Address,
18          new LastMinuteDeal(city, detailed, price));
19      }
20      ... //Other patterns as requested by customer
21  }
```

2.2.2.2 TaskJava

Overview Fischer *et al.* [FMM07, JFM06] proposed *Tasks* as a new programming model for organizing event-driven programs. *Tasks* is a variant of cooperative multithreading and allows each logical control flow to be modularized in a traditional manner. TaskJava, an extension of Java, is developed to instantiate *Tasks*. A *task*, like a thread, encapsulates an independent work-unit in a method called `run`. As a result, the logical control flow of each work-unit is preserved. Moreover, at the same time, tasks can be automatically implemented by the compiler in an event-driven style by using non-blocking Input/Output (I/O) libraries. Programmers needs to use a special `wait` primitive provided by the language instead of registering a callback with each I/O call. A `wait` causes the current task to block until one of a specified set of events occurs. The compilation strategy used with TaskJava is a restricted form of Continuation Passing Style (CPS). There are some advantages of TaskJava compared to existing cooperative multitasking systems:

- TaskJava is scheduler-independent. TaskJava programs can be "linked" to any scheduler that provides the semantics of the `wait` primitive.

- TaskJava properly handles the interactions of `wait` with Java language features including raised exceptions and method overriding.

Programming with TaskJava This part presents a simple Web server in TaskJava (adapted from [JFM06]). The server processes simple one-line Hypertext Transfer Protocol (HTTP) requests for files, and then the contents are sent to the client. We assume that

each connection between the client and the server, which can be considered as a *channel*, supports the following kinds of event:

- READ_RDY_EVT: This event signals that there is incoming data available to be read from the channel.
- WRITE_RDY_EVT: This event signals that the associated channel is ready for writing.
- ACPT_RDY_EVT: This event signals that an accept request is available on the associated channel.
- ERR_EVT: This event signals that an error has occurred on the channel.

The event objects are instances of an `Event` class that is defined as follows (see Listing 2.2.6):

Listing 2.2.6 *The Event class in TaskJava.*

```
public class Event {
    //Enumeration of event identifiers
    //provided by the scheduler
    public static final int READ_RDY_EVT = 1;
    public static final int WRITE_RDY_EVT = 2;
    public static final int ACPT_RDY_EVT = 3;
    public static final int ERR_EVT = 4;
    //type () returns which kind of event occurred
    public int type() { ... }
    //getError() returns the event's error, or null if no
    //error occurred
    public IOException getError() { ... }
}
```

Listing 2.2.7 shows a task-based nonblocking I/O library and Listing 2.2.8 shows an implementation of the Web server in TaskJava (adapted from [JFM06]).

Listing 2.2.7 *A task-based nonblocking I/O library.*

```
public class TaskIO {
    public static class Reader {
     ...
    public async String readLine() throws IOException {
```

```
5         String line;
6         //Keep reading until we finish a line
7         do {
8             Event event = wait(ch, Event.READ_RDY_EVT,
9              Event.ERR_EVT);
10            if (event.type() == Event.READ_RDY_EVT) {
11                ch.read(cbuf);
12                line = scanChars();
13            }
14            else {
15                assert event.type() == Event.ERR_EVT;
16                throw event.getError();
17            }
18        } while (line==null);
19        return line;
20    }
21    public static async void write(CharChannel ch,
22     CharBuffer data) throws IOException {
23        while (data.hasRemaining()) {
24            Event event = wait(ch, Event.WRITE_RDY_EVT,
25             Event.ERR_EVT);
26            if (event.type() == Event.READ_RDY_EVT)
27                ch.write(data);
28            else {
29                assert event.type() == Event.ERR_EVT;
30                throw event.getError();
31            }
32        }
33    }
34    public static CharChannel accept(CharChannel ch)
35        throws IOException { ... }
36    ...
37}
```

Listing 2.2.8 *A Web server written in TaskJava.*

```
1public class TaskWebServer {
2    public static start(CharChannel acceptCh) {
3        spawn AcceptTask(acceptCh);
4    }
5    public static class AcceptTask implements Task {
6        CharChannel acceptCh;
7        AcceptTask(CharChannel acceptCh) {
8            this.acceptCh = acceptCh;
9        }
10       public void run() {
11           try {
12               while (true) {
13                   CharChannel ch = TaskIO.accept(acceptCh);
14                   spawn RequestTask(ch);
```

```
15                  }
16              } catch (IOException e) {
17                  ... print error message to log ...
18                  acceptCh.close();
19              }
20          }
21      }
22      public class RequestTask implements Task {
23          private CharChannel ch;
24          public RequestTask(CharChannel ch) { this.ch = ch; }
25          public void run() {
26              TaskIO.Reader rdr = new TaskIO.Reader(ch);
27              try {
28                  while (true) { //Main request loop
29                      String filename
30                       = parseRequest(rdr.readLine());
31                      charBuffer sendData = readFile(filename);
32                      TaskIO.write(ch, sendData);
33                  }
34              } catch (IOException e) {
35                  ...
36              }
37          }
38      }
39  }
```

2.2.2.3 EventScript

Overview EventScript [Coh07, CK08, Coh08] is an event-processing language based on using regular expressions with actions. The underlying structure of an EventScript program is a regular expression for the sequence of events expected to be received. When a sequence of events (input events) matches a particular pattern, the program emits corresponding output events. To illustrate the basic structure of EventScript, let us consider a simple example that averages readings from three sensors `S1`, `S2`, and `S3` as shown in Listing 2.2.9 (adapted from [CK08]). Whenever a new reading is received from any one of these sensors, the event-processing agent emits an event containing the average value of three readings.

Listing 2.2.9 *A simple EventScript program.*

```
1 in double S1Input, double S2Input, double S3Input
2 out double Average { v1=0.0; v2=0.0; v3=0.0;} (
3     ((S1Input(v1)) | (S2Input(v2)) | (S3Input(v3)))
```

```
4          { !>Average((v1+v2+v3)/3.0);}
5 )*
```

In EventScript, there are two types of events. The first one is input events and the other one is output events. In the example above, `S1Input`, `S2Input` and `S3Input` are input events while `Average` is an output event. Each of these events carries a value with data type *double*. All actions are defined within curly braces. Each action can be an assignment or an emission. For example, the assignment actions at line 2 initializes the variables `v1`, `v2`, and `v3`. These variables are used to keep values read/received from three sensors (not shown here). The emit action (denoted by the symbol `!>`) at line 4 is applied each time a new data is acquired. Two regular expressions are applied in this simple example. The larger one is of the form (...)* (line 5), which indicates repeated instances of the regular subexpression inside the parentheses. Using regular expressions in EventScript will be detailed below.

Event Markers and Event Classification In EventScript, the event marker contains the event name and its associated variables. For example, the event marker `S1Input(v1)` is a placeholder for an occurrence of an event named `S1Input`. Besides, there is a wildcard event marker (`.`) that matches any event.

Additionally to the input events declared by event declarations, EventScript comes with two built-in events markers related to time. The event marker `elapse[x TimeUnit](t)` is matched when `x` time unit(s) have passed since the previous event. Time unit can be in days, hours, minutes, seconds, milliseconds, or microseconds. The time at which the match occurs is stored in the variable `t`. The event marker `arrive[time]` is matched at a specific time and date.

Regular Expression Operators Regular expressions can be combined to make a larger one by using regular operators such as $*, |$ and sequences, as well as parentheses. Table 2.1 (adapted from [CK08]) gives the list of all regular expression operators which can be used in EventScript.

Regular expression	**Event sequence matched**
event marker	a single named input event, or the arrival of a particular time
R^*	zero or more consecutive subsequences, each matching R
$R+$	one or more consecutive subsequences, each matching R
$R?$	either the empty sequence or any sequence matching R
$R_1\|...\|R_n$	any sequence matching at least one of $R_1, ..., R_n$
$R_1, ..., R_n$	n consecutive subsequences matching $R_1, ..., R_n$ in that order
R_1 & R_2	any sequence matching both R_1 and R_2
$R_1 - R_2$	any sequence matching R_1 but not matching R_2
$R\{i\}$	i consecutive subsequences, each matching R
$R\{i, j\}$	from i to j consecutive subsequences, each matching R
$R\{i, \}$	i or more consecutive subsequences, each matching R

Table 2.1: Regular expression operators in EventScript.

2.2.2.4 Urbiscript

Overview UrbiScript [Bai05, BDHN10, Gos13] is a scripting language primarily designed for robotics. It is a dynamic, prototype-based, object-oriented scripting language. Moreover, it supports and emphasizes parallel and event-based programming, which are popular paradigms in robotics, by providing core primitives and language constructs [Gos11].

Programming with UrbiScript UrbiScript enables programmers to define events that can be caught with the `at` and `whenever` constructs. Programmers can create events by instantiating the `Event` prototype. Listing 2.2.10 displays a sample UrbiScript session illustrating events (adpated from [Gos13]):

Listing 2.2.10 *A simple event scenario in Urbiscript.*

```
1 var myEvent = Event.new;
2 [00000000] Event_0xb5579008
3 at (myEvent?)
4 echo("ping");
```

```
5 myEvent!;
6 [00000000] *** ping
7 //Events work well with parallelism
8 myEvent! & myEvent!;
9 [00000000] *** ping
10 [00000000] *** ping
```

In this example, we first define an event called `myEvent` at line 1. Then we define the corresponding action when we receive it (lines 3-4). For testing, we try to emit `myEvent` at line 5 (the exclamation mark `!` denotes an event emission). The result can be seen at line 6, which is a simple print of the string `ping`. If we emit `myEvent` two times (line 8), the action also will happen twice (lines 9-10).

Another example is shown in Listing 2.2.11 (adpated from [Gos13]).

Listing 2.2.11 *`at` and `whenever` constructs in Urbiscript.*

```
1 var myEvent = Event.new;
2 [00000000] Event_0xb558a588
3 whenever (myEvent?)
4 {
5     echo("ping␣(whenever)")|
6     sleep(200ms)
7 };
8 at (myEvent?)
9 {
10     echo("ping␣(at)")|
11     sleep(200ms)
12 };
13 //Emit myEvent for 0.3 second.
14 myEvent! ~ 300ms;
15 [00000000] *** ping (whenever)
16 [00000100] *** ping (whenever)
17 [00000000] *** ping (at)
```

Both `at` and `whenever` have the same behavior on punctual events. However, if a program emit an event for a given duration, `whenever` will keep triggering for this duration in contrast to `at`. In other words, `at` seems like `if` and `whenever` seems like a `while` loop. In UrbiScript, events behave very much like "channels": listeners use `at` or `whenever`, and producers use `!`. The messages can include a payload, i.e. something sent in the "message". The event then behaves very much like the identifier of the message type. To send/catch the payload, programmers just pass arguments to `!` and `?` as shown in Listing 2.2.12 (adpated

from [Gos13]).

Listing 2.2.12 *Emitting and receiving events in Urbiscript.*

```
var event = Event.new;
[00000000] Event_0x0
at (event?(var payload))
echo("received: " + payload)
onleave
echo("had received: " + payload);
event!(1);
[00000008] *** received: 1
[00000009] *** had received: 1
event!(["string", 124]);
[00000010] *** received: ["string", 124]
[00000011] *** had received: ["string", 124]
```

Like functions, events have an `arity`, i.e. they depend on the number of arguments. For instance, `at (event?(arg))` will only match emissions whose payload contain exactly one argument, e.g. `event!(arg)`. In case that the event handlers do not specify their `arity` (i.e. without parentheses), they will match event emissions of any `arity` as shown in Listing 2.2.13. Since they have no arguments, the payload is ignored.

Listing 2.2.13 *The match of event emissions in Urbiscript (continuation of Listing 2.2.12, adpated from [Gos13]).*

```
at (event?)
echo("received an event")
onleave
echo("had received an event");
event!;
[00000014] *** received an event
[00000015] *** had received an event
event!(1);
[00000016] *** received: 1
[00000017] *** had received: 1
[00000018] *** received an event
[00000019] *** had received an event
event!(1, 2);
[00000020] *** received an event
[00000021] *** had received an event
```

2.2.3 Overall Evaluation

Table 2.2 shows the summary of all event-based languages listed in this section. The overall evaluation for each of them is shown below:

- **EventJava**. EventJava is an extension of Java, whose ultimate aim is to support event correlation. However, the event concept in EventJava is mixed with other notations and constructs. Event handlers/callbacks are called explicitly, through a call to the method. In essence, event in EventJava is an asynchronous method. Besides, the events have to be generated "by hand" similar to the observer design pattern [GHJV95, ST04].

- **TaskJava**. TaskJava is an extension of Java to support event-based style. In TaskJava, events are blocking due to the `wait` primitive. However, the notion of event is mixed with other notions. In other words, events are not defined and handled in an intuitive and straightforward way. Besides, there are no event's parameters and callbacks. Lastly, the capabilities of TaskJava are limited to I/O operations.

- **EventScript**. EventScript allows programmers to define and combine events by using regular expressions. Therefore, one of its limitations is the lack of support for conditions expressed by logical expressions, as well as events that are not directly linked to variables' modifications. Moreover, EventScript does not support concurrent programming, which means that events cannot run in parallel.

- **Urbiscript**. UrbiScript is a programming language for robotics with features syntactic support for concurrency and event-based programming. In essence, events in Urbiscript are treated like functions which have arguments and that asynchronously serve to exchange data. Some limitations of UrbiScript are the lacks of support for synchronization among events and of dynamically changing events' behavior at runtime. Moreover, there is no static typing in UrbiScript. Therefore, typing errors may occur while executing. This is particularly a problem when developing reactive and self-adaptive software with high software quality constraints.

	EventJava	**TaskJava**	**EventScript**	**UrbiScript**
Events are declared explicitly?	No	No	Yes	Yes
Support for timing events	No	No	Yes	Yes
Support for concurrency	Need to use Java multi-threading	Need to use Java multi-threading	No	Yes
Support for synchro-nization among events	No	No	No	No
Events can be recon-figured at runtime?	No	No	No	No
Static typing	Yes	Yes	Yes	No
Support for developing user-defined events in other languages	No	No	No	Yes
Possible applications	Many	Many	Many	Mostly robotics

Table 2.2: Summary about the features of state-of-the-art event-based programming languages.

Although these languages or extensions can be used to write event-driven applications, their capabilities are limited as summarized in Table 2.2. A recurrent missing feature is that these languages do not support event synchronization among events, which is essential when there are many events running in parallel (if at all supported by the language). Furthermore, they also do not allow a dynamic modification of events' behavior when running to adapt to changes in the environment, i.e. each event always follows a predefined behavior although this cannot be the best strategy in every situation.

2.3 Rule-Based Programming

2.3.1 Overview

Rule based programming is inspired from the observation of conditioned reflexes of animals and humans, which are trained associations *Condition* ⇒ *Action* [Cro11]. A rule combines two parts: a premise (or a constraint) and a corresponding action [BS84, Abd01]. The action is only executed when the constraint is satisfied. In other words, a rule is a kind of instruction or command that can be applied in certain situations and is a lot like the traditional `if-then` statement of classical programming languages [Hil03]. However, two main differences exist:

- There is a more involved "pattern-matching" than the normal evaluation of `true/false`, which makes rules resemble pattern-matching constructs in functional programming languages.

- Rules do not belong to the usual program control flow, which means that they can be applied at any point of the program.

In practice, rule-based style is often used for reactive intelligence, expert systems and autonomous systems [BS84, Has04, GGGT09]. Rules are also good for programming context-aware systems since they can map contexts to actions [Kru09, RPS05, Lok06, LBM10, WMSC11, NYS+05].

In the next sections, we introduce several representative rule-based programming languages: CLIPS, Jess, Prolog, OPS-2000, and Tom. There are also other rule-based languages that interested readers may have a look at such as ELAN [KK04] and Drools [Bro09, Ama12]. Since the underlying mechanism of rule-based paradigm is not too complicated, it seems that current rule-based languages do not differ from each other too much. The major variation, besides the syntaxes, is the rule evaluation order. For example, the rules may be evaluated sequentially or in an arbitrary order.

2.3.2 Rule-Based Programming Languages

2.3.2.1 CLIPS

CLIPS [CLI13] [4] is a productive development and delivery expert system tool which provides a complete environment for the construction of rule-based and/or object-based expert systems. In CLIPS, rules are used to represent heuristics, or "rules of thumb", which specify a set of actions to be performed for a given situation. A rule is composed of an antecedent (the left-hand side (LHS)) and a consequence (the right-hand side (RHS)).

The antecedent of a rule is a set of related conditions, which need to be satisfied before the rule can be applied. The conditions of a rule are satisfied based on the evaluation for existence or absence of specified facts in the fact-list or specified instances of user-defined classes in the instance-list. For example, a pattern is one type of condition that can be specified. Generally, patterns consist of a set of restrictions that are used to determine which facts or objects satisfy the conditions indicated by the patterns.

The consequence of a rule is a set of actions which will be invoked when the rule is applicable. When more than one rule is applicable, the inference engine in CLIPS uses a conflict resolution strategy to select which rule should have its action executed. Next, the action inside the selected rule is executed and this may affect the list of applicable rules. Then the inference engine selects another rule and executes its actions. This process continues until there is no more applicable rule in a system.

Listing 2.3.1 *Writing rules in CLIPS.*

```
(defrule unsatisfactory-engine-state-conclusions ""
    (working-state engine unsatisfactory)
    =>
    (assert (charge-state battery charged))
    (assert (rotation-state engine rotates)))
```

Programmers may define facts and rules as shown with a simple CLIPS example in Listing 2.3.1 (taken from [Pol10]). This rule is a part of an auto repair expert system. In case that the main car engine's state is not as desired, we need to check whether the battery is

[4] CLIPS is an acronym for "C Language Integrated Production System".

charged and the rotary engine rotates or not. To understand more about CLIPS, please refer to its online documentation [Gia07] or to [GR98].

2.3.2.2 Jess

Jess, a descendant of CLIPS, is a general-purpose rule engine and scripting language written in Java [FH12, Hil03]. Jess allows the definition of rules by using the following functions and constructs:

- `defrule`: defines a new rule.
- `ppdefrule`: pretty-prints a rule.
- `run`: begins firing activated rules from the agenda (for scheduling purpose).
- `undefrule`: deletes a rule.
- `watch rules`: prints a diagnostic when a rule is fired.
- `watch activations`: prints a diagnostic when a rule is activated.

Programmers may define Jess rules by using the `defrule` construct. For instance, they may define a very simple rule (adapted from [Hil03]):

```
Jess> (defrule null-rule
    "A␣rule␣that␣does␣nothing"
    =>
    )
TRUE
```

In this example, the symbol `null-rule` is the name of the rule. In Jess, if a rule named `my-rule` already exists, and programmers define another rule also named `my-rule`, the first version is overwritten (programmers may call the `undefrule` function to delete a rule explicitly). Then the name is followed by an optional description string that expresses the purpose of the rule. The symbol `=>` separates the rule's LHS (i.e. the `if` part) from its RHS (i.e. the `then` part). In our example, the rule `null-rule` has no conditions on its LHS and no actions on its RHS. In other words, it will always be executed and will not do anything.

Let us consider a more complex rule (adapted from [Hil03]):

```
Jess> (defrule change-baby-if-wet
    "If␣baby␣is␣wet,␣change␣its␣diaper."
    ?wet <- (baby-is-wet)
    =>
    (change-baby)
    (retract ?wet))
TRUE
```

Similarly, this rule has two parts. The LHS includes a simple pattern `(baby-is-wet)`. The RHS contains two function calls, to `change-baby` (its details are ignored here) and `retract` (as shown later in Listing 2.3.2 at lines 25-26).

In Jess, programmers may add or remove facts in a "knowledge base". A simple but complete Jess session is shown in Listing 2.3.2 (adapted from [Hil03]).

Listing 2.3.2 *A simple but complete Jess session.*

```
Jess> (clear)
TRUE
Jess> (watch all)
TRUE
Jess> (reset)
==> f-0 (MAIN::initial-fact)
TRUE
Jess> (deffunction change-baby ()
(printout t "Baby␣is␣now␣dry" crlf))
TRUE
Jess> (defrule change-baby-if-wet
"If␣baby␣is␣wet,␣change␣its␣diaper."
?wet <- (baby-is-wet)
=>
(change-baby)
(retract ?wet))
MAIN::change-baby-if-wet: +1+1+1+t
TRUE
Jess> (assert (baby-is-wet))
==> f-1 (MAIN::baby-is-wet)
==> Activation: MAIN::change-baby-if-wet : f-1
<Fact-1>
Jess> (run)
FIRE 1 MAIN::change-baby-if-wet f-1
Baby is now dry
<== f-1 (MAIN::baby-is-wet)
1
```

2.3.2.3 Prolog

Prolog [Llo84] [5] has its roots in first-order logic. Prolog is a declarative programming language: the program is expressed in terms of relations, represented as assertions (facts) and rules under the form of `then` clauses. When running a query over these relations, a computation can be initiated.

To illustrate Prolog, let us consider a rule system which identifies various species of birds (adapted from [Amz10]). Normally, if we apply the `if-then` format, a rule for identifying a particular albatross is:

```
if
family is albatross and color is white
then
bird is laysan_albatross
```

In Prolog, programmers may define the same rule as follows (adapted from [Amz10]):

```
bird(laysan_albatross) :- family(albatross), color(white).
```

The real syntax of Prolog is `then-if`, and the normal RHS and LHS are inverted.

To understand more about Prolog, please refer to [CM03, Sto89, Spi08].

2.3.2.4 OPS-2000

OPS-2000 [NAS12] is a hybrid, interactive, rule-based and object-based, software development environment developed by NASA, a part of the United States government that is in charge of U.S. science and technology related to airplanes or space. There are two primary types of rules in OPS-2000: forward chaining and backward chaining. Forward chaining rules can be further divided into two more groups: those having threshold expressions, and those that do not. A forward chaining rule has the generic syntax given below:

```
defrule <name> {
    <LHS> => <RHS>
}
```

[5] Prolog is an acronym for "PROgramming LOGic".

A rule's LHS consists of pattern logic. If the condition in LHS is satisfied (matched), an activation of the rule is placed into the agenda. Then the conflict resolution algorithm will be applied to determine which agenda entry is to be fired. Similar to other rule-based languages, when a rule is fired, its action (i.e. RHS) is executed. For example, the rule below (adapted from [NAS12]) expresses that if the monkey is at position 15, then issue a command to move it to position 16.

```
defrule RuleOne {
    (The monkey is at position 15) =>
        assert("Move monkey to position 16");
}
```

A backward chaining rule has the generic syntax given below:

```
defrule <name> : bc {
    <LHS> <= <RHS>
}
```

The LHS is a single goal pattern, and the RHS is a subgoal list and/or pattern logic. The RHS pattern logic is pattern-matched exactly as if it was a forward chaining rule's LHS.

2.3.2.5 Tom

Tom [INR13a] is a pattern matching compiler developed at INRIA, the French national research institution focusing on information and computer science and technology. This language extension based on Java is particularly well-suited for programming various transformations on trees/terms and XML[6]-based documents. Its design follows INRIA's research on the semantics and the efficient compilation of rule based languages.

To illustrate the definition and use of rules in Tom, let us consider a sorted list module (as shown in Listing 2.3.3), which maintains invariants with rule-based hooks (adapted from [INR13b]):

Listing 2.3.3 *A sorted list with Tom.*

[6]XML is an acronym for "Extensible Markup Language".

```
1 import rules.sortedlist.types.*;
2 public class Rules {
3     %gom {
4         module sortedlist
5         imports int
6         abstract syntax
7         Integers = sorted(int*)
8         /* We define a normalizing rewrite system for the
9          module (only one rule here) */
10        module sortedlist:rules() {
11            /* Every time a term with 'sorted' as an head
12             symbol is constructed, the following conditional
13             rewrite rule is applied, hence ensuring an
14             invariant on the lists */
15            sorted(x,y,t*) -> sorted(y,x,t*) if y <= x
16        }
17    }
18    public static void main(String[] args) {
19        //Testing the module with several examples
20        Integers l1 = `sorted(7,5,3,1,9);
21        Integers l2 = `sorted(8,4,6,2,0);
22        Integers l3 = `sorted(l1*,10,l2*);
23        System.out.println(l1 + "\n" + l2 + "\n" + l3);
24    }
25 }
```

The rule at line 15 indicates that when `y` is not greater than `x`, we will swap them. When running with Tom, the example result, which displays three sorted lists, is shown below:

```
user@host$ tom Sort.t && javac Sort.java && java Sort
sorted(1,3,5,7,9)
sorted(0,2,4,6,8)
sorted(0,1,2,3,4,5,6,7,8,9,10)
```

2.4 Conclusion

In this chapter, we presented the state-of-the-art in the fields of context-aware pervasive computing, event-based and rule-based programming styles. Many frameworks, tools, services, and paradigms have been put forward to support context-awareness. Since event-based and rule-based programming styles are suitable for capturing and handling changes in the environment, several authors consider using them as a good solution to develop

context-aware reactive applications.

Current programming language support for context-awareness still has some limitations. We take into account their disadvantages and overcome these limitations by developing INI. The ultimate goal is helping programmers to write context-aware applications more intuitively and straightforwardly. In the next chapter, we will introduce INI in more details, including its syntax, features, and how it supports multithreading, context-awareness, and dynamic behavior adaptation.

Chapter 3

Introducing INI

In this chapter, we present major features of INI, mostly on how events and rules can be applied. For events, besides basic uses, we also introduce some advanced uses such as event synchronization, event reconfiguration, and the combination of an event with a guard. Finally, several examples of INI will be shown to illustrate the main concepts and ideas of our language.

3.1 Motivation

After considering the drawbacks and disadvantages of current programming language support to context-aware adaptive and reactive computing (see Section 2.1.5 (page 39) and Table 2.2 (page 54)), we believe that the new one should meet the succeeding foremost requirements:

- The language should allow a well-defined mechanism to recognize and handle both occurred internal and external changes when programs are running. In other words, the system should be equipped with a robust and flexible manner to capture all events that may happen. Programmers may adjust some parameters to tune this process and then receive returned results. Besides, the language syntax should be intuitive, easy to use and major concepts can be defined clearly.

- Reactions may run in parallel to exploit multithreading and improve the performance and responsiveness of the programs. Indeed, parallelism is must-have for the modern

and future computing [FM11].

- Reactive behavior may be dynamically customized at runtime to adapt to requirements/environments/conditions.
- The language should come with a strong type checking engine to prevent any data type conflict.
- A support must exist for programmers to apply straightforwardly state-of-the-art verification and validation techniques to check properties and constraints that need to be satisfied by their programs.

Other essential demands involve code elegance, simplicity, conciseness, readability and support for the integration of built-in external modules written in other classical languages like Java or C/C++. To sum up, our ultimate goal is to make context-aware programs easier to write, develop and check.

3.2 Features

3.2.1 Overview

INI [1] is a programming language developed by ourselves at ISEP [ISE13, LP12] since 2010 (initially designed by Renaud Pawlak and later enhanced by Truong-Giang Le), with an interpreter that runs on Java Virtual Machine (JVM). Although INI is linked to Java, its syntax and semantics are not Java's ones. We select the JVM on the grounds that it is powerful and flexible to provide a natural home for programming languages other than Java [EV12, Gho10].

Each INI program contains functions, which combine event expressions, logical expressions (used to specify the trigger conditions) and the action (lists of statements) bound to them. Note that the scope of all variables is the whole function. The syntax does not require typing but all type conflicts are prohibited at compile time (type inference is used). INI supports basic features as in other languages like arithmetic/logical expressions,

[1] Originally, INI stands for "INI is Not ISEP", as an allusion to the famous self-referencing definition "GNU is Not Unix". Besides, the name INI of our language also represents the drawing of an event (the N) passing between two interfaces (the Is).

map/list/set expressions, the `case` statement (similar to the `if` statement). Besides, with our language, programmers may use some advanced features such as set selection expressions, user-defined types and type pattern matching. All statements end with a new line instead of a semicolon. Furthermore, INI allows programmers to use Java objects through binding. This takes advantage of rich existing functions and Application Programming Interfaces (APIs) in Java in case that they do not want to develop new ones. Therefore, INI inherits from all existing Java libraries and can be easily extended in Java. For instance, many Java APIs for I/O operations (e.g. working with data streams, files, and directories) may be reused, just by declaring a header, to save time and effort. In Chapter 3 and Chapter 4, we will discuss those INI's features in details.

The major characteristic of INI is that it combines both event-based and rule-based programming styles. Either events or rules can be defined independently or in combination. Events in INI run in parallel either synchronously and asynchronously, depending on the constraints. Programmers may develop their own events in other languages such as Java or C/C++. Additionally, event's behavior may be dynamically customized during execution time. In the following sections, we will explain on how programmers may define and use rules and events.

3.2.2 Rules in INI

A rule in INI consists of a logical expression and a corresponding action. When the logical expression part of a rule is evaluated to `true`, the action is invoked. To illustrate rules in INI, let us show a simple example that implements some sort of an infinite ping-pong game between two players (Listing 3.2.1).

Listing 3.2.1 *A ping-pong program written in INI.*

```
1 function ping_pong() {
2     @init() {
3       v=1
4     }
5     v == 2 {
6       println("pong")
7       v = 1
8     }
9     v == 1 {
```

```
10          println("ping")
11          v = 2
12      }
13  }
```

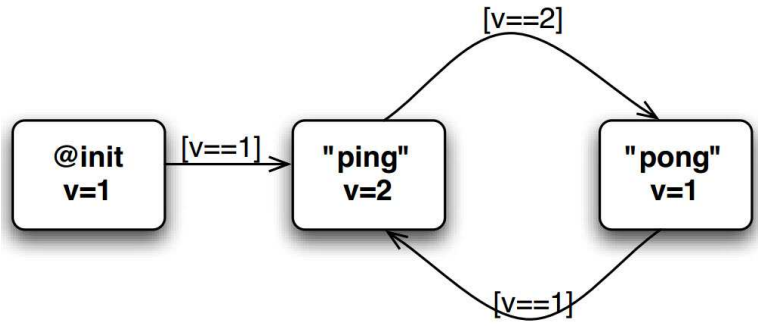


Figure 3.1: The `ping-pong` function state machine.

When entering that program, INI first evaluates the `@init` event at line 2 that initializes the variable `v` to 1 (see more in Section 3.2.3.1, page 68). Then, INI sequentially tries out all the rules of the function in their declaration order. Here, the guard at line 5 evaluates to `false`, but the one at line 9 evaluates to `true`, thus triggering the evaluation of the action that prints out `"ping"` (line 10) and sets `v` to 2 (line 11). After that, the function will not terminate before all the guards evaluate to `false`. So, since line 5 now evaluates to `true`, INI evaluates the rule that prints out `"pong"` (line 6) and resets `v` to 1 (line 7). This action obviously triggers again the other rule, thus starting the endless ping-pong game over again. Note that the rule declaration order is not important here. Any other rule order would give the same result. That's because the guards are *disjoint*. The `ping-pong` example corresponds to the state machine shown in Figure 3.1.

Another example of rules is using INI for calculating the factorial. In mathematics, the factorial of a non-negative integer `n`, denoted by `n!`, is the product of all positive integers less than or equal to `n`. Listing 3.2.2 shows two versions of the factorial function written in INI, one in rule-based style and the remaining is in recursive style.

On the left, we can see the rule-based version that defines:

1. An `@init` event (see more in Section 3.2.3.1, page 68) that defines and initializes a variable to store the result (`f`) and another variable to store the current integer to

multiply (`i`).

2. A rule that multiplies the result by the current integer `i` and increments the latter, guarded by the condition that `i` is lower than or equal to the number `n` we want to calculate the factorial of. Due to the INI execution semantics, this rule will continue to apply until `i<=n` becomes `false`, which eventually happens since `i` is incremented at each rule execution (`i++` expression). Once `i` is larger than `n`, there is no rule to be executed anymore in the function.

3. A termination `@end` event (see more in Section 3.2.3.1, page 68) that returns the calculated result `f`.

Listing 3.2.2 *Calculating N Factorial with INI.*

```
function fac(n) {
    @init() {
        f=1
        i=1
    }
    i <= n {
        f=f*i++
    }
    @end() {
        return f
    }
}
```

```
function fac(n) {
    n==1 {
        return 1
    }
    n > 1 {
        return n*fac(n-1)
    }
}
```

The recursive version on the right-hand side is quite simpler since the factorial calculation is easy to define recursively. Thus, we can define a rule that recursively uses `fac(n-1)` to calculate `fac(n)` for `n>1`, and a terminal rule for `n==1`, that simply returns 1 (the factorial value for `fac(1)`) for terminating the recursion. Note that each rule causes the function to end immediately because of the `return` statement.

Besides logical expressions, programmers may also use regular expressions or pattern matching expressions to specify the guard that need to be satisfied before the action is executed (see Section 4.1.8 of Chapter 4, page 94).

3.2.3 Events in INI

3.2.3.1 Built-in Events

In INI, we support all kinds of events (e.g. timing events, state changes, physical events) described in Section 2.2, page 39. Event callback handlers are declared in the body of functions and are raised, by default asynchronously, every time the event occurs (the execution of the handler represents the event instance). By convention, an event in INI starts with `@` and the corresponding event kind. It takes input and output parameters. Input parameters are configuration parameters to tune the event execution. Output parameters are variable names that are filled in with values when the event callback is called. They can be considered as the measured characteristic of the event occurrence. Those variables, as well as any INI variable, enjoy a lexical scope in the function's body. Both types of parameters are optional. Moreover, an event kind can also be optionally bound to an id, so that other parts of the program can refer to it (e.g. for event synchronization or event reconfiguration). For example, an event `e:@sampleEvent[iParameter1 = v1](oParameter1,oParameter2)` has `e` as id, one input parameter named `iParameter1` and its corresponding value `v1`, along with two output parameters called `oParameter1` and `oParameter2`, which are variables that can be used anywhere in the whole function body, as specified a few lines above.

Built-in event kind	**Meaning**
`@init()`	used to initialize variables, when a function starts.
`@end()`	triggered when no event handler runs, and when the function is about to return.
`@every[time:Integer]()`	occurs periodically, as specified by its input parameter (in milliseconds).
`@update[variable:T] (oldValue:T, newValue:T)`	invoked when the given variable's value changes during execution.
`@cron[pattern:String]()`	used to trigger an action, based on the CRON pattern indicated by its input parameter.

Table 3.1: Some built-in events in INI.

To allow programmers to write code more easily and conveniently, INI comes with some common built-in event kinds (as listed and described in Table 3.1). The `@init` event is invoked at the beginning during the evaluation of a function. The `@end` event is executed

before a function terminates. The `@update` event is used to monitor the internal changes of a system, i.e. when variables are modified. Besides, there are two timing events:

- `@every` event: repeats an action after a specific time.
- `@cron` event: CRON [2] is a task scheduler that allows the concise scheduling of a repetitive task within a single (and simple) CRON pattern [NSHW10]. A UNIX crontab-like pattern is a string split in five space separated parts, composed of minutes sub-pattern, hours sub-pattern, days of month sub-pattern, months sub-pattern, and days of week sub-pattern.

To illustrate built-in events in INI, the code in Listing 3.2.3 creates an `@every` instance called `e`, which increments `v` every second. The `@update` instance `u` triggers the action (event handler) that prints out the variable `v`'s value when it changes, i.e. every second.

Listing 3.2.3 *Using built-in events in INI.*

```
function main() {
    @init() {
        v = 0
    }
    e: @every[time=1000]() {
        v = v + 1
    }
    u: @update[variable=v](oldv, newv) {
        println("v␣has␣changed␣from␣" + oldv + "␣to␣" + newv)
    }
}
```

When running, the output result will look like that:

```
v has changed from 0 to 1
v has changed from 1 to 2
...
```

Note that the input parameter names are mandatory, while output parameter names are optional (i.e. programmers may choose the names they want). For example:

[2] CRON stands for "Command Run ON".

- `@every[time=1000]()`: good.
- `@every[x=1000]()`: bad.
- `@every[time="one thousand"]()`: bad.
- `@update[variable=x](y,z)`: good.

3.2.3.2 User-Defined Events

Overview Programmers may also implement user-defined events (in Java or in C/C++), and then integrate them to their INI programs. By developing custom events, one can process data which are captured by sensors. To illustrate user-defined events in INI, let us consider a program which uses sensors to capture and collect weather and climate data like humidity, temperature, wind speed, rainfall, etc. In our program, we can define separate events to handle these tasks as shown in Listing 3.2.4. For instance, we can define an event `@humidityMonitoring` to observe the humidity level periodically. This event has one input parameter named `humPeriod` that sets the periodicity of the checks (time unit is in hours). Besides, it has one output parameter named *humidity* to indicate the current humidity. Inside this event, depending on the value of the current humidity, we can define several corresponding actions such as warning when the humidity is too high by using the `case` construct (described later in Section 4.1.3.3, Chapter 4, page 89). Other events can be defined in a similar way. All events in our program run in parallel so that it can handle multiple tasks at one time. The next part will detail how programmers may write user-defined events in INI.

Listing 3.2.4 *An INI program monitoring weather and climate data.*

```
function main() {
    h:@humidityMonitoring[humPeriod = 1](humidity) {
        case {
            humidity > ... {...}
            default {...}
        }
    }
    t:@temperatureMonitoring[tempPeriod = 2](temperature) {
        ...
    }
```

```
    ...
}
```

Implementing User-Defined Events To exemplify on how programmers may write user-defined events, we discuss a sample INI program, which uses a video camera to detect the movement of a ball, gets its positions in space periodically, and saves the collected position data in a Comma-separated Values (CSV) file. (a similar event will be used later in Section 6.2, page 160). To do so, we first need to define a new event kind to detect the ball and send its position to the program when detected. Our event will have one input parameter called `period`. This parameter is applied to set how long the event should sleep between two image detections (time unit in milliseconds). Besides, we will have three output parameters `(r,x,y)`, which are the radius and coordinates of the detected ball in the captured image. To develop this built-in event in Java, we need to subclass the `ini.event.Event` class to define the behavior of our new event kind as shown in Listing 3.2.5.

Listing 3.2.5 *Writing user-defined events.*

```
public class BallDetection extends ini.event.Event {
    Thread ballDetectionThread;
    @Override public void eval(final IniEval eval) {
        (ballDetectionThread = new Thread() {
            @Override public void run() {
                ...
                do {
                    try {
                        //Sleep as long as the configuration
                        //indicates
                        sleep(getInContext().get("period").
                         getNumber().longValue());
                        //Use OpenCV to detect the ball
                         ...
                        //Write data to output parameters
                        variables.put(outParameters.get(0), r);
                        variables.put(outParameters.get(1), x);
                        variables.put(outParameters.get(2), y);
                        //Execute the event action
                        execute(eval,variables);
                    } catch (Exception e) {...}
                } while (!checkTerminated());
            }
```

```
24          }).start();
25      }
26      @Override public void terminate() {...}
27  }
```

In the `BallDetection` class, the method `eval` will be upcalled by the INI evaluator when the program uses our event. First, it creates a thread that sleeps accordingly to the event configuration as indicated by the input parameter `period` (lines 11-12), and then detects the ball using OpenCV, a library for programming computer vision [Gar13] (line 13). Next, the results are written in output parameters to be passed to the INI program (lines 16-18). The event-triggered action passed as a parameter at line 3, also called *event thread*, is executed at line 20 using the `execute` method provided by the APIs in INI, which by default runs asynchronously. Finally, the method `terminate` is overridden to stop the event (line 26): INI upcalls this method when the program exits or forces the event to terminate.

Listing 3.2.6 *A sample INI program with a user-defined event.*

```
1 @ballDetection[period:Integer](Float, Integer, Integer)
2  => "ini.ext.events.BallDetection"
3 function main() {
4      @init() {
5          f = file("ballData.csv")
6          case {
7              !file_exists(f) { create_file(f) }
8          }
9      }
10     //Use our event get notified for ball detection
11     b:@ballDetection[period = 1000](r,x,y){
12         fwriteln(f,to_string(time())+","+r+","+x+","+y)
13     }
14 }
```

In Listing 3.2.6, we write the actual INI program, which binds our Java class to the `@ballDetection` event kind at lines 1-2 (in other examples of INI in this paper that use user-defined events, we make this kind of declaration implicitly). In the `@init` event, we define a variable `f`, which indicates the CSV file we want to store data after collecting the ball positions over time. If the file does not exist, we create it (line 7). Since in INI, all variables enjoy a lexical scope within the function where they are defined, in particular, `f` can be accessed at line 12 inside the event `@ballDetection`, as well as at lines 5 and 7.

In our program, the `@ballDetection` event is triggered periodically, i.e. each second. If a ball is detected, we write the data to the file (line 12), including the time when the ball was detected and its position.

Programmers may also develop their own events in C/C++ and then port them to Java by using the library JNA [3] [Wal13] or JavaCPP [Aud13]. In essence, writing user-defined events in INI includes the following steps:

- Step 1: Subclass the `ini.event.Event` class.
- Step 2: Declare a thread initialized for running the event each time it is triggered.
- Step 3: Override the method `run` that is applied to handle the event's task. In this part, programmers may use JNA or JavaCPP to invoke C/C++ code.
- Step 4: Bind the values calculated in Step 3 to output parameters.
- Step 5: Override the method `terminate` to handle the desired action when the event is terminated.

3.2.3.3 Event Synchronization and Reconfiguration

By default, except for the `@init` and `@end` events (see Table 3.1, page 68), all INI events are executed asynchronously. However, in some scenarios, a given event `e0` may want to synchronize on other events `e1,..., eN`. It means that the synchronizing event `e0` must wait for all threads running a handler corresponding to the target events to be terminated before running. For instance, when `e0` affects the actions defined inside the other events, we need to apply the synchronization mechanism. Note that one of the target events can also be synchronized with `e0`. Cross-synchronization of events means that their executions are mutually exclusive.

Furthermore, programmers may apply INI to handle changes happening in the environment though the event-reconfiguration mechanism. Essentially, event reconfiguration consists of modifying the values of the event's input parameters. Programmers can invoke the built-in function `reconfigure_event(eventId, [inputParam1 = value1, inputParam2`

[3] JNA stands for "Java Native Access".

`= value2,...])` in order to reconfigure their events. Moreover, we also allow programmers to stop and restart events with the built-in functions `stop_event([eventId1, eventId2, ...])` and `restart_event([eventId1,eventId2, ...])`. Typically, it is required to stop an event before reconfiguring it.

Let us now consider our ball-detection example of Section 3.2.3.2 runs in an embedded environment where the power is supplied by a battery. One way to take into account this new constraint is to adapt the data-collection period to the power level. First, we add a new user-defined event kind called `@powerAlarm`, which notifies the program each time the power level passes a given threshold both ways, when charging or discharging. This event has one output parameter named `currentLevel`, which tells us the current power level (either lower, equal or greater than the threshold). When the program in Listing 3.2.7 is running, if it detects that the power-level is lower than 50%, it stops the event `b:@ballDetection` (line 12), then changes the value of its input parameter (i.e., `period`) to 100000 (line 13), and finally restarts it (line 14). Conversely, if the power goes over the threshold, the value for the parameter `period` is set again to 1000 (lines 18-23). The event `@powerAlarm` is synchronized on the event `b:@ballDetection` as specified by `$(b)` at line 8 in order to avoid unfinished detection jobs to terminate cleanly before applying reconfiguration.

Listing 3.2.7 *Dynamic adaptation through event reconfiguration.*

```
1  ...
2  threshold = 50
3  setLow = false
4  b:@ballDetection[period = 1000](r,x,y){
5      ...
6  }
7  //Adapt the ball detection period at the 50% threshold
8  $(b) p:@powerAlarm(currentLevel) {
9      case {
10         //Augment the period to save energy
11         currentLevel < threshold && !setLow{
12             stop_event(b)
13             reconfigure_event(b, [period = 100000])
14             restart_event(b)
15             setLow = true
16         }
17         //Recover default settings
18         currentLevel > threshold && setLow {
```

```
19              stop_event(b)
20              reconfigure_event(b, [period = 1000])
21              restart_event(b)
22              setLow = false
23          }
24      }
25  }
```

To understand more about using events in INI, interested readers may refer to INI Language Reference Documentation [LP12].

3.2.3.4 Using Events with Guards

In INI, events and guards may be used in combination. A guard (guard condition) is a logical expression, that is used to express the requirements that need to be satisfied before an event can be executed.

For example, considering again the weather monitoring system (see Listing 3.2.4, page 70), if we want the event `@temperatureMonitoring` to be executed only when the humidity is higher than some threshold:

```
@temperatureMonitoring[tempPeriod = 2](temperature)
 humidity >... {
    //Do an action ...
}
```

3.3 Programming with INI

In this part, we show several examples to illustrate the capabilities of INI and also to help programmers get familiar with our language, especially with the definitions and uses of rules and events.

3.3.1 Implementing a Sort Function

With INI, implementing a sort function can be done by taking advantage of *set expressions*. A set expression allows to select an arbitrary object within a set. This selects an object which satisfies some criteria given in the second part of the expression. Set expressions are used in the guard part of a rule so that the guard is evaluated to `true` only if an object that matches the given criterion can be found in the set. In the following

paragraphs, two kinds of sorting algorithms, namely bubble sort and quicksort, will be mentioned for illustration.

Bubble sort In Listing 3.3.1, to implement the `sort1` function (based on the bubble sort or sinking sort algorithm [CLRS09, Lev12]), we simply select all the indexes `i` in the `s` list so that `s[i]` is greater than `s[i+1]` (line 2). When this rule matches (satisfied), it simply swaps `s[i]` and `s[i+1]`, so that the list will eventually be sorted when the rule cannot be applied anymore. In that case, the `@end` event is triggered and the function returns the sorted list.

Listing 3.3.1 *Implementing bubble sort in INI.*

```
1 function sort1(s) {
2     i of [0..size(s)-2] | s[i] > s[i+1] {
3         swap(s[i],s[i+1])
4     }
5     @end() {
6         return s
7     }
8 }
```

Programmers can also use explicit indexes to iterate on the list to be sorted, leading to more classical code. The following function implements a basic bubble sort with INI (as shown in Listing 3.3.2). The iteration is done with two rules: one which swaps the current elements (lines 7-11), and one which does not (lines 12-14). A `swap` boolean flag is used to know if there was a swap during the iteration or not. If `swap` is true once the end of the list is reach, we set back the index `i` to `0` so that the iteration rules are applied all over again (lines 15-18).

Listing 3.3.2 *A classical approach for implementing bubble sort in INI.*

```
1 function sort2(s) {
2     @init() {
3         i = 0
4         swap = false
5         size = size(s)-1
6     }
7     i < size && s[i] > s[i+1] {
8         swap(s[i],s[i+1])
9         swap = true
```

```
10          i++
11      }
12      i < size && s[i] <= s[i+1] {
13          i++
14      }
15      i==size && swap {
16          swap = false
17          i = 0
18      }
19      @end() {
20          return s
21      }
22  }
```

When compared with the corresponding program written in Tom (see Listing 2.3.3, page 60), our implementation seems to be easier to understand when all rules can be declared and used explicitly. In Tom, the mechanism on how a conditional rewrite rule is applied (i.e. `sorted(x,y,t*) -> sorted(y,x,t*) if y <= x`) is done implicitly.

Quicksort The two implementations for bubble sort above have a similar complexity of $\mathcal{O}(n^2)$. In order to get a better complexity, we can for instance implement the quicksort (or partition-exchange sort) algorithm [CLRS09, Lev12]. The divide-and-conquer strategy is applied in quicksort, which involves the following steps:

1. Step 1: Choose a pivot value that can be any element but normally is the middle one (especially for longer partitions).

2. Step 2: Rearrange the array such that all elements smaller than the pivot value go to the left part, and all elements greater than the pivot value go to the right part. If the value of an element is equal to the pivot value, this element can stay in either the left or the right part.

3. Step 3: Apply recursively these above steps for both the left and the right parts until no more part can be applied.

Our INI program implementing this algorithm with the use of rule-based style is shown in Listing 3.3.3.

Listing 3.3.3 *Implementing the quicksort algorithm in INI.*

```
function quicksort(s,lo,hi){
    hi>lo && !done{
        p = partition(s,lo,hi,lo)
        //Applying on both the left and right parts
        quicksort(s, lo, p-1)
        quicksort(s, p+1, hi)
        done = true
    }
    @end() {
        return s
    }
}

function partition(s,lo,hi,pivotIndex){
    @init() {
        pivotValue = s[pivotIndex]
        swap(s[pivotIndex],s[hi])
        index=lo
        i=lo
    }
    i<hi && s[i]<=pivotValue {
        swap(s[i],s[index])
        index++
        i++
    }
    i<hi && s[i]>pivotValue {
        i++
    }
    @end() {
        swap(s[hi], s[index])
        return index
    }
}
```

This algorithm is more complex to understand but it performs very well for middle-sized lists (the average case performance is $\mathcal{O}(n \log n)$). The interesting characteristic to point out here is that the quicksort function is recursive. Hence, INI allows altogether rule-based, event-based and more classical programming styles. Programmers may use one or the other depending on the type of problem they need to solve and implementation requirements.

3.3.2 N-queens Problem

In chess, a queen can move flexibly: horizontally, vertically, or diagonally. The standard 8-queens problem is to place eight queens on a chess board (with eight rows and eight

columns) in such a way that no queen can capture any of the others (one possible solution is shown in Figure 3.2).

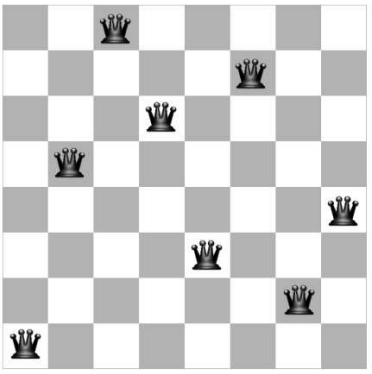


Figure 3.2: A solution to the 8-queens problems.

A more general problem is to consider an N by N "chess board" and how to place N queens on such a board so that no two queens attack each other (i.e. they are not in the same row or in the same column or in the same diagonal) [Cha03, Moh08, Mun09]. The N-queens problem is a classic puzzle in computer science and also regarded as a great programming exercise. In Listing 3.3.4, we show how programmers may solve this problem by applying rules in INI.

Listing 3.3.4 *An INI program solving the N-queens problem.*

```
/**
 * This program finds all the solutions to the N-queens puzzle
 */
function main() {
    @init() {
        pos = []
        find_solutions(pos,8)
    }
    @end() {
        println("\nEnd.")
    }
}

function check_solution(pos) {
    @init() {
        print(".")
    }
    i,j of [0..size(pos)-1] | j>i &&
     (pos[i]==pos[j] || (i + pos[i] == j + pos[j])
```

```
20        || (i - pos[i] == j - pos[j])) {
21            return false
22        }
23        @end() {
24            return true
25        }
26    }
27
28    function find_solutions(pos,size) {
29        @init() {
30            line = size(pos)
31            pos[line] = 0
32        }
33        ok = check_solution(pos) {
34            find_solutions(copy(pos), size)
35        }
36        ok && size(pos) == size {
37            println("\nFound␣solution:")
38            print_pos(pos,size)
39        }
40        pos[line] == size-1 {
41            return
42        }
43        pos[line] < size-1 {
44            pos[line]++
45        }
46    }
47
48    function print_pos(pos,size) {
49        @init() {
50            i = 0
51        }
52        i < size {
53            print_line(pos,i,size)
54            i++
55        }
56    }
57
58    function print_line(pos,line,size) {
59        @init() {
60            i = 0
61        }
62        i<size && line<size(pos) && i==pos[line] {
63            print("Q␣")
64            i++
65        }
66        i<size && (line >= size(pos) || i!=pos[line])  {
67            print("-␣")
68            i++
69        }
70        @end() {
```

```
71          println("")
72      }
73  }
```

In our program, the vector `pos` is used to indicate the columns of all queens that are already placed on the chess board. For example, `[1,3,6]` means that we have three queens at three positions: `[0,1]` (first row, second column), `[1,3]` (second row, fourth column), and `[2,6]` (third row, seventh column). The function `find_solutions` is applied to find the solutions. Inside this function, we invoke the function `check_solution` to verify whether the new placed queen violates the requirements regarding all existing queens on the board or not. The situation that one queen can capture another queen is expressed at lines 18-20. If no violation is found (i.e. the function `check_solution` returns `true`), we try to place the next queen (a rule at lines 33-35). When all `N` queens are placed appropriately on the board, this means that one solution is found (a rule at lines 36-39). The two rules at lines 40-42 and lines 43-45 are used to test all possible positions of each queen to find all possible solutions. Finally, the two functions `print_pos` and `print_line` display the solution on the screen, in which each position of a queen is represented by a letter `Q`.

3.3.3 An Online Ordering System

Let us consider an online ordering system shown in Listing 3.3.5. Let's assume that our system only allows at most 100 orders per day. In our program, we use two rules. The first rule at lines 7-13 is applied to accept new order requests and count the number of orders which have been accepted. The second rule at lines 14-18 is applied to stop providing the service when the number of orders has reached the limit. During execution, while the variable `allowOrderOnline` is `true`, the action inside the first rule will be executed. Then with the second rule, when the variable `allowOrderOnline` is still `true` and the number of orders is over 100, the variable `allowOrderOnline` is assigned to `false`. The reason why we combine two logical expressions, namely `allowOrderOnline` and `numOfOrder > 100`, is that we do not want to invoke this assignment many times.

Listing 3.3.5 *An online ordering system written in INI.*

```
1 function main() {
```

```
2     @init() {
3         orderId = 0
4         allowOrderOnline = true
5         numOfOrder = 0
6     }
7     allowOrderOnline {
8         //The function accept_new_order_request() accepts,
9         //keeps information related to a customer's request,
10        //and returns the id of the order.
11        orderId = accept_new_order_request()
12        numOfOrder++
13    }
14    allowOrderOnline && numOfOrder > 100 {
15        //Reach the threshold, do not allow online
16        //ordering anymore
17        allowOrderOnline = false
18    }
19    @update[variable=orderId](oldOrderId, newOrderId) {
20        //Get information related to the new order and
21        //handle it ...
22    }
23    @every[time=24*3600*1000]() {
24        numOfOrder = 0
25        allowOrderOnline = true
26    }
27}
28...
```

Along with rules, several events are used in our program. The event `@init` is used to initialize necessary variables. During execution, when there is a new order, the value of the variable `orderID` is changed, and therefore, the event `@update` at lines 19-22 will be invoked. The action of this event (instance) handles the customer's request, and it runs in a separate, newly created, thread. In other words, we allow many orders simultaneously. Moreover, the event `@every` at lines 23-26 is applied to reset two program's variables (`numOfOrder` and `allowOrderOnline`) after each 24 hours in order to provide again the online ordering service.

3.3.4 An Automatic Lighting Control System

Let us consider an automatic lighting control system in a corridor (displayed in Listing 3.3.6, adapted from [LHM+12]). In our program, there is one user-defined event called `@motionDetection`, which is applied to detect movement also by using the library OpenCV. This event has two input parameters named `mode` and `period`, which are set to point out

whether we apply a simple algorithm or an advanced one for detecting motion and the period for checking (time unit is in seconds). Whenever a motion is detected, our program will turn on the lights if they were turned off before (lines 9-12). The event `@every` at lines 15-17 is applied to compute how much time passed without any motion. If there is no movement within fifteen minutes and the current lights state is on, the rule at lines 18-21 will be invoked to turn off the lights to save energy. The event `@update` (lines 22-24) can be used to invoke some desired actions when the light status is changed (i.e. the lights are turned on or turned off).

Listing 3.3.6 *An automatic lighting control system written in INI.*

```
1  function main() {
2      @init() {
3          lightOn = false
4          timeWithNoMotion = 0
5      }
6      @motionDetection[mode = "simple", period = 30]() {
7          timeWithNoMotion = 0
8          case {
9              !lightOn {
10                 //Turn on the lights
11                 lightOn = true
12             }
13         }
14     }
15     @every[time = 60000]() {
16         timeWithNoMotion++
17     }
18     timeWithNoMotion > 15 && lightOn {
19         //Turn off the lights
20         lightOn = false
21     }
22     @update[variable=lightOn]() {
23         //Do a desired action ...
24     }
25 }
```

3.3.5 An Intelligent Virtual Personal Assistant

Listing 3.3.7 shows an intelligent virtual personal assistant written in INI, which can recognize voice commands from users and then do appropriate actions (adapted from [LHM+12]). One of the most interesting features of our program is that it can detect

who is using it (based on face detection), in order to adjust the voice recognition process with regard to his or her maternal language, tones and accents. This data is previously collected and stored in a database during a speech training procedure. As a result, the accuracy of voice recognition will be improved.

Listing 3.3.7 *An intelligent virtual personal assistant written in INI.*

```
function main() {
    @init() {
        defaultId = 1
        currentId = 1
    }
    $(v) f:@faceRecognition(isKnown, recognizedId) {
        case {
            //A familiar person is detected, then change
            //current settings to new settings
            isKnown && recognizedId != currentId {
                currentId = recognizedId
                stop_event(v)
                reconfigure_event(v,[userId=currentId])
                restart_event(v)
            }
            //A stranger is detected, then use default settings
            !isKnown {
                ...
            }
        }
    }
    v:@voiceRecognition[userId=defaultId](voiceCommandString) {
        case {
            //Execute an action based on the voice command
            voiceCommandString ~ regexp(...) {
                //Do action 1
            }
            ...
            default {
                //Do a default action ...
            }
        }
    }
}
```

The `@init` is used to initialize some variables as usual. Besides, there are two user-defined events in our program. The event `@faceRecognition` (identified by `f`) is applied to detect a human face (also by using the library OpenCV). This event has one output parameter called `recognizedId`, which is used to indicate the corresponding id (if exists)

of the user. If a new face is detected, we stop the event `@voiceRecognition` (identified by `v`), reconfigure it depending on whether the user has been recognized or not, and restart the event `v` in order to improve the performance and precision of voice recognition process. The event `@voiceRecognition` is used to recognize user's voice. It has one input parameter specifying the id of the user, which is applied in order to tune the recognition pattern. Moreover, there is one output parameter called `voiceCommandString`, which is the returned/recognized spoken sentence. At line 25, we match the user's command against a regular expression (see more in Section 4.1.8, page 94) to guess its meaning (by using the match operator $\sim$), and then do a suitable action. The event `f` should be synchronized on the event `v`, which means that if there is a current running thread for `v`, `f` has to wait before it can be executed. The synchronization is necessary since we want to avoid unfinished voice recognition jobs to be stopped.

3.4 Comparison between INI and Other Languages

When compared with other language-based approaches used to develop context-aware applications (see Chapter 2), working with INI has several major characteristics as listed below:

- All context information categories may be captured and handled through different types of events. Events in INI can be used to monitor time perception, time passing, internal changes of the systems, data collected from sensors, etc. Besides, rules in INI are evaluated sequentially and may be used for reaction purpose when some conditions are satisfied.

- Both events and rules can be defined intuitively for reactive purpose. The notions of events and rules do not mix with other ones. Besides, events come with input and output parameters to tune the execution and bring returned results.

- INI allows programmers to write user-defined events in other languages such as Java or C/C++. Each user-defined event can be viewed as an external independent module/component that may be applied in many INI programs.

- INI supports concurrency. Events in INI run in parallel either synchronously or asynchronously. This improves the performance of INI programs when they need to deal with multitasking. For example, INI may collect and handle multiple values read from different kinds of sensors at one time.

- INI supports dynamic adaptation at runtime. The behavior of an event can be reconfigured to handle better varied situations happening in the environment. This approach can be considered as one aspect of "smart computing".

Chapter 4

Formalizing INI

In this chapter, we introduce the syntax for basic INI's elements such as functions, events, and rules (Section 4.1). Furthermore, we define an operational semantics for INI (Section 4.2). Our approach tries to explain how an INI program works regarding event and rule managing and triggering mechanisms and their effects on the program state. Since event synchronization is one of the main concepts in INI, we clarify this by detailing the synchronization algorithm (Section 4.3).

4.1 Syntax

4.1.1 Expressions

INI has some basic expressions as in other languages:

- Arithmetic expression: an expression that results in a numeric value with two kinds of numerical values, which are integers (whole numbers), and real or floating point numbers (numbers containing a decimal point). Arithmetic expressions can be formed by connecting literals (the number itself, written with digits) and variables with one of the arithmetic operators $\{+, -, *, /, \%\}$
- Logical expression: are formed of variable accesses, function invocations, and literals (e.g. strings, numbers, etc.), composed together with classical logical operators (`&&` (and), `||` (or), `!` (not)) and/or comparison operators (`==`, `!=`, `>`, `>=`, `<`, `<=`).
- List expression: an ordered collection (also known as a sequence) like [1,2,3,4,5] or

[“red”,“green”,“blue”].

Other more complex expressions like regular expressions, type pattern matching expressions, guard expressions, event expressions, will be presented in the next parts.

4.1.2 Types and Type Declarations

INI supports basic types: *Numeric* (*Double*, *Float*, *Long*, *Int*, *Byte*), *Char*, *String*, *Boolean*.

Besides, INI also has an internal `Void` type for functions returning no values. We then have some rules for literals, which can directly be inferred as resolved types:

- String literals such as `“abc”` will be typed as *String*.
- Character literals such as `‘a’` will be typed as *Char*.
- Float literals such as `3.14` will be typed as *Float*.
- Integers such as `1` will be typed as *Int*.
- `true` and `false` literals will be typed as *Boolean*.

Programmers can also define variables of type map, list, and set (will be explained in details in Section 4.1.7, page 92). Some simple examples to illustrate these types are shown below:

```
l = [1,2,3,4,5] //l is a list
//Declare a map that maps names to ages
m["Giang"] = 29
m["Truong"] = 26
//s is a set that contains all the integers
//between 0 and 10 (bounds included)
s = [0..10]
```

Besides, programmers may declare user-defined types and use pattern matching operators to select correct types for instance variables. Those features will be clarified later in Section 5.2 (page 118).

4.1.3 Statements

Statements in INI can be variable assignments, function invocations, `case` statements, and `return` statements.

4.1.3.1 Assignment

Programmers may declare variables without declaring explicitly their types. For example:

```
i = 1 //i is an integer
j = 2.0 //j is a float
str = "Hello World" //str is a string
```

4.1.3.2 Function Invocation

Functions takes parameters which have a unique name within the function scope. Parameters are passed by reference and not by value. If programmers wants to pass by value, it can be done using the `copy` built-in function. For instance, the function:

```
function f(a,b,c) { ... }
```

can be invoked with `f(1,2,"abc")`, if `f` expects two integers and one string. Parameters can have default values. For example, the `b` and `c` parameters may have default values:

```
function f(a,b=0,c="") { ... }
```

In that case, the parameter values are optional when invoked. For instance, `f(2,1)` invokes `f` with an empty string for `c`.

4.1.3.3 Case Statement

In INI, a `case` statement allows programmers to handle different possibilities related to execution conditions. During evaluation, the action corresponding to the first satisfied condition is executed. If there is no such condition, the default action (if exists) will run. Its syntax is shown as follows:

```
case {
    <logical_expr_1> {
        <statements>
    }
    <logical_expr_2> {
        <statements>
    }
    ...
    default {
        <statements>
    }
}
```

4.1.3.4 Return Statement

A function can return a value by using a `return [<expr>]` statement within a rule's body. If `<expr>` is not defined or if no return statements appear in the function, then the function returns a `Void` value. For instance:

```
1 function f(a,b=0,c="") {
2     ...
3     <guard> {
4         ...
5         return b
6     }
7     ...
8 }
```

The `return` statement at line 5 indicates that the function returns a value of type `Int` (since `b` is an integer, as seen in the parameter's default value at line 1). Note that `return` statements are optional when the function returns no value, but must always be the last statement of a rule.

4.1.4 Function Declarations

Each INI program contains functions, whose bodies combine event expressions, logical expressions (used to specify the condition that trigger the rule) and the actions (lists of statements) bound to them. The scope of any variable is the whole function. A function in INI has the following syntax:

```
function <name>(<parameters>) {
    <logical_expression> { <statements> }
     | <event_expression> { <statements> }
```

```
    | <event_expression> <logical_expression> { <statements> }
}
```

4.1.5 Rules

A rule in INI consists of a logical expression and a corresponding action:

```
<logical_expression> { <statements> }
```

Besides logical expression, INI also defines a match operator ($\sim$) that allows for regular expression matching on strings and for pattern matching, similarly to the match construct in OCaml [Smi06] (see Section 4.1.8, page 94). The logical expression must be evaluated to `true` to allow the evaluation of the rule's action.

4.1.6 Events

4.1.6.1 Declaring Events

The syntax of events is shown below:

```
id:@eventKind[inputParam1=value1, inputParam2=value2, ...]
 (outputParam1, outputParam2, ...)
    { <statements> }
```

in which,

- `id` is the identifier for the event (optional).
- `eventKind` indicate which type of event is applied. For instance, the event can be `@every`, `@update`, or `@ballDetection`.
- `inputParam1`, `inputParam2`... are input parameters' names and `value1`, `value2`... are expressions applied to set values.
- `outputParam1`, `outputParam2`... are output parameters' names.

So we can see that since each event kind may be declared several times, an event may be named with an identifier in order to distinguish. Regarding each declared event, there also may be several instances for it that run in parallel at one time. For example, in the following code, some instances for both `e1` and `e2` can run together:

```
1 e1:@every[time=2000]() {...}
2 e2:@every[time=5000]() {...}
```

The syntax corresponding to an event `e0` that is synchronized on other events `e1,...,eN`, is:

```
$(e1,e2,...,eN) e0:@eventKind[...](...) { <statements> }
```

4.1.6.2 Operating on Events

Programmers can invoke the built-in function `reconfigure_event(eventId, [inputParam1=value1,inputParam2=value2,...])` in order to adjust their events' behavior. Moreover, we also allow programmers to stop and restart events with the built-in functions `stop_event([eventId1, eventId2, ...])` and `restart_event([eventId1,eventId2, ...])`. Typically, those steps are needed when reconfiguring an event.

4.1.6.3 Using Events with Guards

An event expression can be used with a guard (expressed as logical expressions):

```
<event_expression> <logical_expression> { <statements> }
```

4.1.7 Maps, Lists, and Sets

4.1.7.1 Map Definition and Access

Maps are natively supported by INI. A map is automatically defined when accessed through the map access expression that uses square brackets: `map[key]`. Keys and values within maps are of any type, but shall all be of the same type for a given map. For instance the following statement list is valid:

```
1 m["key1"] = 1
2 m["key2"] = 2
3 println(m["key2"])
```

On the other hand, the succeeding statement list is not valid because the first line initializes `m` to be a map of integer values accessed through string keys and:

- The key is an integer at line 4, so there will be a typing error.

- The value is a string at line 5, so there will be a typing error.

```
4 m[1] = 2
5 m["key2"] = "test"
```

Note that the size of a map (i.e. the number of elements in it) can be retrieved with the `size` built-in function. Emptying a map or removing an entry is done with the `clear` function (see more in [LP12]).

4.1.7.2 List Definition and Access

In INI, a list is simply a map where keys are consecutive integers, starting from 0. For example, if `l=[1,8,6,7,5]` then `l[3]` is 7.

4.1.7.3 Integer Sets

To allow easy iteration on lists, INI provides a constructor for sets of integers. A set of integers is defined with two bounds: `[min..max]`. For instance, the set that contains all the integers between `0` and `10` (bounds included) is written as `[0..10]`.

4.1.7.4 Set Selection Expressions

Set can be used in set selection expressions that allows programmers to select elements in a set and bind their values to local variables. A selection condition must be used to select the elements upon a given criteria. For instance, to select two integers `i` and `j` that are contained within 0 and 10, so that `i` $<$ `j`, one can write the following set selection expression:

```
i, j of [0..10] | i < j
```

The bounded values can be expressed as arithmetical expressions, like:

```
l = [1,2,3,4,5]
i of [0+1..size(l)-2] | s[i] > s[i+1]
```

After each iteration for checking the index and the corresponding condition, INI will evaluate successive rules in the program and come back for the next iteration later.

Set selection expressions must be used in guards. The difference between a set selection guard and a regular guard is that the former one will be checked until all the possible values

have been picked from the set. In other words, the guard will stop matching only once none of the set elements fits the selection condition.

In INI, set selection expressions can also be used to select instances of user-defined types as shown in Section 5.2.1.3, page 120.

4.1.8 Regular Expressions

In order to match strings in a concise way, INI provides a match operator ∼, which can match a string against a regular expression [Fri06] and bind matching groups (if any) to INI variables (based on Java regular expressions). For instance:

`"a b c"` ∼ `regexp("(.) (b) (.)",v1,v2,v3)` will match and be evaluated to `true`. Since there are three groups (between parentheses), the three match sub-results will be bound to the given variables `v1`, `v2`, and `v3` with the leftmost-outermost strategy. Thus, once the matching is done, we will have `v1="a"`, `v2="b"`, and `v3="c"`. For more information on regular expressions as used in INI, please read the Javadoc for the `java.regexp.Pattern` class [Ora13c] and refer to [Wat05, Fri06, GL09].

To illustrate regular expressions in INI, let us consider a simple example:

```
1 function greetings(sentence) {
2     sentence ~ regexp("Hello␣(.*)",name)
3      || sentence ~ regexp("Hi␣(.*)",name) {
4         println("Hello␣to␣"+name)
5         return
6     }
7     sentence ~ regexp("Bye␣(.*)",name)
8      || sentence ~ regexp("See␣you␣(.*)",name) {
9         println("Bye␣to␣"+name)
10        return
11    }
12 }
```

This function matches the given sentence to determine who it is said hello or bye to. Typically, the invocation `greetings("See you Giang")` will print out `"Bye to Giang"`. Note the use of the `return` statements at lines 5 and 10 to ensure that rules are applied once at best.

4.1.9 Binding to Java Objects

INI only provides a minimal set of built-in functions. For all other functions, one can bind new functions to Java APIs. Thus, to use Java objects from INI, programmers just need to define bindings from INI functions to Java constructors, methods or fields. The binding syntax is the following:

```
<name>(<types>) -> <type> => "string1", "string2"
```

This binding declares a new function named `<name>` that takes parameters typed with the given comma-separated type list (`<types>`) and returns a typed result. The corresponding Java element that will be used when invoking the function is defined thanks to the two strings following the `=>` binding operator, where `string1` is the target Java class fully-qualified name, and `string2` is one of the following:

- The target field name (belonging to the class).
- The target method name (belonging to the class) followed by `(..)` to indicate that it is a method (and not a field).
- `new(..)` used to indicate that the target is a constructor of the class.

It is not needed to specify if the Java method or field is static, since INI will determine it automatically depending on the parameter types of the function. Non-static members will require to pass an instance of the type of the target class as the first parameter.

For instance, the following code defines two bound functions to call the classical `System.out.println(..)` method in Java. The `out()` function binds to the static `System.out` field, and the `java_println()` function binds to the `Writer.println(String)` non-static method.

```
out()->Writer => "java.lang.System", "out"
java_println(Writer,String)->Void =>
    "java.io.Writer", "println(..)"
```

Programmers can then invoke both functions as shown below, which are well-typed thanks to the binding declarations.

```
java_println(out(),"hello␣Java")
```

4.1.10 Imports

Since all the functions should not be defined within a single file, INI programs can start with a list of import clauses. For example:

```
import "ini/lib_examples/lib_io.ini"
```

Imports allow the definition and use of function libraries. In particular, it is recommended to define bindings (see the previous part, Section 4.1.9) within external files and to import them when required.

4.2 Operational Semantics

4.2.1 Introduction to Operational Semantics

It is essential to define a formal semantics for a programming language since this provides [Bak06] the users with:

- An unambiguous description of the effect of a program (i.e. giving the meaning to a program in a mathematical rigorous way).
- A yardstick for implementation.
- A basis for program analysis and synthesis.

Operational semantics is one of the three common approaches to semantics, along with denotational semantics and axiomatic semantics [Win93]. The purpose is to map one program context (initial state) to another one (final state) that we take to be the result of the program. Each program context is described by using some representative symbols to illustrate program elements. Operational semantics can be classified into two main categories:

- Big-step operational semantics or natural semantics [Kah87] (a.k.a. relational semantics or evaluation semantics): it specifies the entire transition from the initial state to the final value. For example, the notion $\langle expression, state \rangle \Downarrow v$ states that the *expression* within the *state* is eventually evaluated to v.

- Small-step operational semantics or structural operational semantics (popularized by Plotkin [Plo81, Plo04], a.k.a. transitional semantics or reduction semantics): it specifies the transition of a program one step at a time. There is a set of rule that can be applied to the initial state so that we can reach the final state, which is defined to be a state in which no transition applies. In other words, we evaluate a program step-by-step. For instance, the notion $\langle expression_1, state_1\rangle \Rightarrow \langle expression_2, state_2\rangle$ states that the $expression_1$ within the $state_1$ will be evaluated to the $expression_2$ within the $state_2$.

Some pros and cons of those two presentations are listed in Table 4.1. To understand more about operational semantics, interested readers may refer to [Hen90, NN92, Win93, SK95].

	Pros	**Cons**
Big-step	Easier to write since we may skip intermediate simple steps.	More abstract and intuitive, but cannot express complex behavior.
Small-step	More expressive. Can model complex behavior such as looping, concurrency, etc.	In some cases, it is too complicated to express the transitions.

Table 4.1: Comparison between big-step and small-step operational semantics.

4.2.2 Operational Semantics for INI

We combine both big-step ($\Downarrow$) and small-step ($\Rightarrow$) operational semantics when defining semantics for INI. Besides, we also have a look at other approaches, which have some common properties [Bro96, NH96a, Wei97a, GMP04, MG08, BBF+12].

4.2.2.1 Structures for INI's Constructs

We use the following syntactic categories and their corresponding meta-variables:

- n will range over numerals
- b will range over boolean values, $b \in B = \{true, false\}$

- x will range over variables
- *expr* will range over basic general expressions
- *ae* will range over arithmetic expressions
- *le* will range over logical expressions
- r will range over pattern matching expressions for regular expressions
- t will range over type pattern matching expressions
- l will range over list expressions
- *cs* will range over the `case` statements
- *cb* will range over the bodies of the `case` statements
- s will range over statements
- ϵ is an empty statement
- *setSel* will range over set selection guards
- *setExpr* will range over set expressions
- g will range over guard expressions
- *eip* will range over expressions for event's input parameters
- *event* will range over event expressions
- f will range over functions
- *body* will range over functions' bodies

The structures of all constructs in INI are defined as follows:

- $expr ::= ae \mid le \mid l$
- $ae ::= n \mid x \mid ae_1$ `nop` $ae_2 \mid f(expr^*)$
  where `nop` ranges over numeral-valued binary operations, `nop` $\in Nop = \{+, -, *, /, \%\}$.

- $le ::= \ true \mid false \mid ae_1$ `cop` $ae_2 \mid le_1$ `bop` $le_2 \mid$ `ubop` $le \mid f(expr^*)$

  where

  - `cop` ranges over boolean-valued numeric comparison operations, `cop` $\in Cop = \{==, !=, >=, <=, >, <\}$.

  - `bop` ranges over boolean-valued binary boolean operations, `bop` $\in Bop = \{\&\&, ||\}$.

  - `ubop` ranges over boolean-valued unary boolean operations, `ubop` $\in Ubop = \{!\}$.

- $l ::= \ ae^* | le^* \mid f(expr^*)$

- $cb ::= \ le \ \{s\} \mid cb_1 \ NL \ cb_2$

- $cs ::= \ case \ \{cb \ \langle default \ \{s\}\rangle\}$

- $s ::= \ x = expr \mid ae$ `unop` $\mid f(expr^*) \mid return \ expr \mid cs \mid s_1 \ NL \ s_2$

  where `unop` ranges over numeral-valued unary operations, `unop` $\in Unop = \{++, --\}$.

- $r := expr \sim regexp(PAT, x_1, ..., x_n)$

  where PAT is a regular expression pattern.

- $t := expr \sim TYPE[]$

  where $TYPE$ is a defined type.

- $eip ::= \ x = expr \mid x_1 = f(x_2^*) \mid eip_1, eip_2$

- $event_{def} ::= \langle \$(event\_id^+)\rangle \ \langle \ event\_id :\rangle \ event\_kind\langle [eip]\rangle(x^*)$

- $setExpr ::= \ [ae_1..ae_2] \mid ALGEBRAIC\_TYPE$

  (see Section 5.2.1.2, page 119 to understand more about algebraic data types).

- $setSel ::= \ x$ of $setExpr \mid le$

  where the notation "|" used here belongs to the syntax, not for the case separator.

- $g ::= \ le \mid setSel \mid event \mid le \ event \mid r \mid t$

- $body ::= \ g\{s\} \mid body_1 \ body_2$

- $f_{def} ::=$ `func_id` $(x^*) \ \{body\}$

We use the following conventions:

- NL denotes a new line.
- Elements inside $\langle\ \rangle$ are optional.
- $expr^*$ is a shorthand for a possibly empty sequence $expr_1, expr_2, ..., expr_n$. Other uses of the notation * have the same meaning.
- $expr^+$ is a shorthand for a sequence $expr_1, expr_2, ..., expr_n$ having at least one element. Other uses of the notation $^+$ have the same meaning.

4.2.2.2 Notations and Symbols

As mentioned earlier, each INI function contains rules and events and the latter run in parallel. As a result, function calls are evaluated concurrently, which we reflect in the semantics, by a two-layered structure: at the top level the semantics consists of $\overrightarrow{\mathcal{F}}$, the global execution context. $\overrightarrow{\mathcal{F}}$ is composed of w function execution contexts (described below) running in parallel $\mathcal{F}_1 \mid\mid \mathcal{F}_2 \mid\mid ... \mid\mid \mathcal{F}_w$. Each one of them evaluates one function call.

We adjoin to those w function evaluation contexts (a.k.a. $\overrightarrow{\mathcal{F}}$) a unique event instance generator $\mathbb{E}$ as illustrated in Figure 4.1. Actually, $\mathbb{E}$ can be considered as a black box. The functionality of $\mathbb{E}$ is just taking the input parameters when registering and reconfiguring the (event) callbacks $\overrightarrow{\mathcal{C}}$ (linked to a particular function execution context F_l), then checking whether an event is executable or not based on its properties (see Section 7). Eventually, if the event instance can be executed, $\mathbb{E}$ calculates and binds values to output parameters, and it sends the callback's body to the corresponding thread pool of F_l for it to be executed in the right context. For example, in Figure 4.1, two event instances have been raised, one for the `@every` event, that was the first event declared by (the function call evaluated in) F_1, and the other one is for the `@ballDetection` event, that was declared by F_l.

For example, in Figure 4.1, two event instances have been raised, one for the `@every` event and the remain is for the `@ballDetection` event.

As illustrated in Figure 4.1, a function execution context F_l is composed of:

- $[R, i, j]$ expresses the evaluation of rules, in which `i` is the index for the rule whose

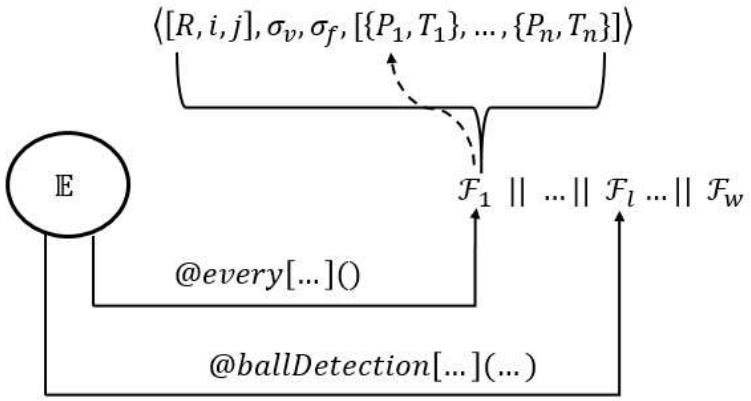


Figure 4.1: The global and function execution contexts and the event generator mechanism in INI.

condition is currently evaluated and `j` is a temporary counter used to specify the number of successive rules (regarding to the rule currently evaluated), which have been evaluated as not applicable (i.e. their concerned conditions are not satisfied).

- A statement replaces $[R, i, j]$ when the body of a concrete rule is evaluated.

- σ_v^l is the variable and parameter context inside the function, which is a `Map`: $VariableName \rightarrow Value$. It includes local variables defined in the function and its own parameters.

- σ_f^l is the function context, which is a `Map`: $FunctionName \rightarrow \Psi$, where Ψ is the set of all 3-tuples abstracting a function: a variable context σ_v, the list R of rules defined inside the function and the list $\overrightarrow{\mathcal{C}}$ of (event) callbacks (see Section 7, page 187). Note that σ_f^l is global and does not depend on the index l of F_l.

- A pair $\{P_k^l, T_k^l\}$ for each event e_k inside the function, in which P_k^l is a thread pool that contains all current execution threads for the instances of e_k (raised by $\mathbb{E}$ and whose execution is not yet finished) and T_k^l is a boolean variable used to indicate whether e_k is terminated or not (e.g. after a call to `stop_event`), in which case only the current threads are allowed to finish running (see Section 7, page 187). Inside the thread pool, instructions are evaluated as the same way as in normal site (note that the event's body includes only statements). We use the notation Θ_P^l to indicate

the set of thread pools $[\{P_1^l, T_1^l\}, \{P_2^l, T_2^l\}, ..., \{P_{n_l}^l, T_{n_l}^l\}]$ for the n_l events defined in the function.

As a shorthand, we let $\Sigma = [\sigma_v, \sigma_f, \Theta_P]$ and we also call it the execution context, by abuse of language. When there will be a need to disambiguate Σ of $\langle \Box, \Sigma \rangle$ (where $\Box$ represents the place for the instruction/rule), we will call the latter as the proper evaluation context. σ_f is a set that contains the function contexts for all functions. The evaluation of an expression `expr` is denoted as $\langle expr, \Sigma \rangle$. To keep the notations simple, we assume that in the sequel $\overrightarrow{\mathcal{F}}$ contains only one execution context and keep $\mathbb{E}$ implicitly when it plays no role (i.e. most of the time). In the following sections, $\Downarrow$ is applied to state that a context is evaluated to a value (big-step semantics), and $\Rightarrow$ is used to express a transition (small-step semantics).

4.2.2.3 Semantics of Statements

The semantics of all statements in INI is defined as follows:

- Assignment

$$\frac{\langle ae,\ \Sigma\rangle\ \Downarrow\ \langle n,\ \sigma'_v,\ \sigma'_f,\ \Theta'_P\rangle}{\langle x = ae,\ \Sigma\rangle\ \Rightarrow\ \langle \epsilon,\ \sigma''_v = \sigma'_v + [x \leftarrow n],\ \sigma'_f,\ \Theta'_P\rangle}$$

$$\frac{\langle le,\ \Sigma\rangle\ \Downarrow\ \langle b,\ \sigma'_v,\ \sigma'_f,\ \Theta'_P\rangle}{\langle x = le,\ \Sigma\rangle\ \Rightarrow\ \langle \epsilon,\ \sigma''_v = \sigma'_v + [x \leftarrow b],\ \sigma'_f,\ \sigma'_P\rangle}$$

- The `case` statement

$$\frac{\langle le,\ \Sigma\rangle\ \Downarrow\ \langle true,\ \Sigma'\rangle}{\langle case\ \{le\ \{s\}\ cb\},\ \Sigma\rangle\ \Rightarrow\ \langle s,\ \Sigma'\rangle}$$

$$\frac{\langle le,\ \Sigma\rangle\ \Downarrow\ \langle false,\ \Sigma'\rangle \quad cb \neq \epsilon}{\langle case\ le\ \{s\}\ cb,\ \Sigma\rangle\ \Rightarrow\ \langle cb,\ \Sigma'\rangle}$$

$$\frac{\langle le,\ \Sigma\rangle\ \Downarrow\ \langle false,\ \Sigma'\rangle}{\langle case\ le\ \{s_1\}\ default\ \{s_2\},\ \Sigma\rangle\ \Rightarrow\ \langle s_2,\ \Sigma'\rangle}$$

- Numeral-valued unary operations

$$\frac{}{\langle x{+}{+},\ \Sigma\rangle\ \Rightarrow\ \langle \epsilon,\ \sigma'_v = \sigma_v + [x \leftarrow (\sigma_v(x) + 1)],\ \sigma_f,\ \Theta_P\rangle}$$

$$\frac{}{\langle x{-}{-},\ \Sigma\rangle\ \Rightarrow\ \langle \epsilon,\ \sigma'_v = \sigma_v + [x \leftarrow (\sigma_v(x) - 1)],\ \sigma_f,\ \Theta_P\rangle}$$

- Sequence

$$\frac{\langle s_1,\ \Sigma\rangle\ \Rightarrow\ \langle \epsilon,\ \Sigma'\rangle \quad \langle s_2,\ \Sigma'\rangle\ \Rightarrow\ \langle \epsilon,\ \Sigma''\rangle}{\langle s_1\ NL\ s_2,\ \Sigma\rangle\ \Rightarrow\ \langle \epsilon,\ \Sigma''\rangle}$$

- The `return` statement

  We define the semantics for the `return` statement invoked inside the function `f`.

$$\frac{\langle expr, \Sigma\rangle \Downarrow \langle v,\ \sigma'_v,\ \sigma'_f,\ \Theta'_P\rangle}{\langle return_f\ expr,\ \Sigma\rangle \Downarrow \langle \bullet,\ \sigma''_v = \sigma'_v + [f \leftarrow v],\ \sigma'_f,\ \Theta'_P\rangle}$$

  where "$\bullet$" indicates the "terminated" state of a function and f is also the name of the returned variable.

4.2.2.4 Semantics of Matching against Regular Expressions

Regular expression matching mechanism in INI is based on the one found in Java (see Section 4.1.8, page 94). There are two cases:

- The matching is successful:

$$\frac{\langle expr, \Sigma\rangle \Downarrow \langle v,\ \sigma'_v,\ \sigma'_f,\ \Theta'_P\rangle \quad v\ REG\ PAT =_{Java} v_1, ..., v_m}{\langle expr \sim regexp(PAT, z_1, ..., z_n),\ \sigma_v,\ \sigma_f,\ \Theta_P\rangle \Downarrow \langle true,\ \sigma''_v,\ \sigma'_f,\ \Theta'_P\rangle}$$

  where $\sigma''_v = \sigma'_v + [z_1 \leftarrow \bot, ..., z_n \leftarrow \bot] + [z_1 \leftarrow v_1, ..., z_{min(n,m)} \leftarrow v_{min(n,m)}]$.

  We can see that if there are too many values ($m > n$), the additional values are ignored. For example, with the expression `"12 345 6789"` $\sim$ `regexp("(\\d+) (\\d+) (\\d+)"),v1,v2)`, `v1` is `12` and `v2` is `345`. Conversely, if there are too many variables ($n > m$), the additional variables are set to `null` (the previous values will be erased if there were some). For instance, with the expression `"Hello INI"` $\sim$ `regexp("Hello:?(INI|ISEP)",v1,v2)`, `v1` is `INI` and `v2` is `null`.

- The matching is unsuccessful:

$$\frac{\langle expr, \Sigma\rangle \Downarrow \langle v, \Sigma'\rangle \quad v\ REG\ PAT =_{Java} \text{matching failure}}{\langle expr \sim regexp(PAT, z_1, ..., z_n),\ \Sigma\rangle \Downarrow \langle false,\ \Sigma'\rangle}$$

4.2.2.5 Semantics of Type Pattern Matching

INI allows programmers to use the match operator with algebraic data types (see more in Section 5.2.1.2, page 119). Basically, it is similar to the regular expression matching (see Section 4.2.2.4, page 103), but does not bind any variable. There are two cases:

- The matching is successful:

$$\frac{\langle expr, \Sigma\rangle \Downarrow \langle v, \Sigma'\rangle \quad v \text{ has been constructed with } TYPE[]}{\langle expr \ \sim \ TYPE[],\ \Sigma\rangle \Downarrow \langle true,\ \Sigma'\rangle}$$

  The premise means that we check in the interpreter's records, that the expression *expr* has been constructed with the type constructor *TYPE*.

- The matching is unsuccessful:

$$\frac{\langle expr, \Sigma\rangle \Downarrow \langle v, \Sigma'\rangle \quad v \text{ has not been constructed with } TYPE[]}{\langle expr \ \sim \ TYPE[],\ \Sigma\rangle \Downarrow \langle false,\ \Sigma'\rangle}$$

4.2.2.6 Semantics of Set Selection Expressions

The use of set selection expressions is explained in Section 4.1.7.4 (see page 93). The operational semantics for the evaluation of a set expression is shown below:

$$\frac{\langle ae_1,\ \Sigma\rangle \Downarrow \langle v_1,\ \Sigma'\rangle \quad \langle ae_2,\ \Sigma'\rangle \Downarrow \langle v_2,\ \Sigma^{(2)}\rangle}{\langle [ae_1..ae_2],\ \Sigma\rangle \Downarrow \langle \{v_1, .., v_2\},\ \Sigma^{(2)}\rangle}$$

where $\{v_1, .., v_2\}$ denotes the set composed of all the integer values v such that $v_1 \leq v \leq v_2$. Note that if $v_1 > v_2$ then the set is empty.

For evaluating a set selection expression, there are two cases:

- At least one index satisfies the bound condition:

$$\frac{\begin{array}{c}\langle le,\ \sigma'_v + [x \leftarrow v_1],\ \sigma_f,\ \Theta'_P\rangle \Downarrow \langle false,\ \Sigma^{(2)}\rangle \\ \langle le\ \sigma_v^{(2)} + [x \leftarrow v_2],\ \sigma_f,\ \Theta_P^{(2)}\rangle \Downarrow \langle false,\ \Sigma^{(3)}\rangle \\ \vdots \\ \langle setExpr, \Sigma\rangle \Downarrow \langle v, \Sigma'\rangle \qquad \langle le,\ \sigma_v^{(i)} + [x \leftarrow v_{i+1}],\ \sigma_f,\ \Theta_P^{(i)}\rangle \Downarrow \langle true,\ \Sigma^{(i+1)}\rangle\end{array}}{\langle x \text{ of } setExpr \mid le,\ \Sigma\rangle \Downarrow \langle true, \sigma_v^{(i+1)} + [x \leftarrow v_{i+1}],\ \sigma_f,\ \Theta_P^{(i+1)}\rangle}$$

  where

  - *setExpr* denotes the set expression.
  - *le* (logical expression) expresses the condition bound to the set selection expression.
  - $v = \{v_1, ..., v_n\}$: all the returned results for the *setExpr* operation.

- No index satisfies the bound condition:

$$\frac{\begin{array}{c}\langle le,\ \sigma'_v + [x \leftarrow v_1],\ \sigma_f,\ \Theta'_P\rangle \Downarrow \langle false,\ \Sigma^{(1)}\rangle \\ \langle le,\ \sigma_v^{(1)} + [x \leftarrow v_2],\ \sigma_f,\ \Theta_P^{(1)}\rangle \Downarrow \langle false,\ \Sigma^{(2)}\rangle \\ \vdots \\ \langle setExpr, \Sigma\rangle \Downarrow \langle v, \Sigma'\rangle \qquad \langle le,\ \sigma_v^{(n-1)} + [x \leftarrow v_n],\ \sigma_f,\ \Theta_P^{(n-1)}\rangle \Downarrow \langle false,\ \Sigma^{(n)}\rangle\end{array}}{\langle x \text{ of } setExpr \mid le, \Sigma\rangle \Downarrow \langle false, \sigma_v^{(n)} + [x \leftarrow x_{init}],\ \sigma_f,\ \Theta_P^{(n)}\rangle}$$

  where x_{init} is $\sigma_v(x)$, the initial value of x.

In INI, we allow the possibility to have multiple variables for set selection expressions, such as:

```
x,y of [0..10] | x < y
```

The semantics for this kind of expression can be defined similarly based on the extension from the case in which there is only one variable. Note that the result returned from the *setExpr* is now the Cartesian product $v = \{v_1, ..., v_n\}^m$ in the m-variables case.

4.2.2.7 Semantics of Rules

Each rule combines a condition expressed as a logical expression *le* and an action in its body (list of statements). When *le* is evaluated to `true`, the action is invoked. We check all rules sequentially from the first one to the last one and loop until no more rules can be applied. The semantics of rules is defined as follows, assuming that r is the total number of rules:

- If the function does not have an `@init` event, the evaluation for rules begins directly with the first rule:

$$\frac{}{\langle \triangleright,\ \Sigma\rangle\ \Rightarrow\ \langle [R,\ 0,\ 0],\ \Sigma\rangle}$$

  where "$\triangleright$" indicates the "starting" evaluation of a function.

- A rule's action is not executed when its condition is evaluated to `false`, then the next rule is checked:

$$\frac{\langle le_i,\ \Sigma\rangle\ \Downarrow\ \langle false,\ \Sigma'\rangle}{\langle [R,\ i,\ j],\ \Sigma\rangle\ \Rightarrow\ \langle [R,\ (i+1) \bmod r,\ j+1],\ \Sigma'\rangle}$$

where "mod" is the modulo operation.

- A rule's action is executed when its condition is evaluated to `true`, then the next rule is checked:

$$\frac{\langle le_i,\ \Sigma\rangle \Downarrow \langle true,\ \Sigma'\rangle \quad \langle body_i,\ \Sigma'\rangle \Rightarrow \langle \epsilon,\ \Sigma''\rangle}{\langle [R,\ i,\ j],\ \Sigma\rangle \Rightarrow \langle [R,\ (i+1) \bmod r,\ 0],\ \Sigma''\rangle}$$

4.2.2.8 Semantics of Events

Each event instance e_{inst} runs in the corresponding thread pool P and involves the following parts:

- The explicit (external) event id *eid* that is shown in an INI program (if used) and to which programmers may refer later (e.g. using functions operating on events like `stop_event`, `reconfigure_event`, `restart_event`). Besides, each event instance comes with an implicit (internal) id mainly for thread pool management (and the programmers do not know about and cannot refer to) to handle the situation where the external id lacks (i.e. an anonymous event). We also call it *eid*, since if the external event id is defined, it can also serve as an internal id. Therefore, for simplicity, we use *eid* for both the internal and the external event ids.
- The event kind (e.g. `@every`, `@ballDetection`, etc.).
- The guard *le* if exists.
- The list L of all event ids on which e_{inst} is synchronized.
- Parameters: the set of event's input parameters' names $IP = ip_1, ip_2, \cdots, ip_p$ and their corresponding values $iv_1, iv_2, \cdots, iv_p$; the set of event's output parameters' names $OP = op_1, op_2, \cdots, op_q$ and their corresponding values $ov_1, ov_2, \cdots, ov_q$.
- The (still unexecuted) event action EA, which combines sequential statements. This is the event handler that is in charge of properly reacting to the event occurrences.
- The boolean value $T_{e_{inst}.eid}$ that is used to indicate whether the event is terminated or not. By default, $T_{e_{inst}.eid}$ is `false`. Calling `stop_event` or `restart_event` on $e_{inst}.eid$ sets $T_{inst.eid}$ to `true` or `false` respectively.

All the components except $T_{e_{inst}.eid}$ form the callback $\mathcal{C}_{e_{inst}.eid}$. We use RE to indicate the list of runnable event instances inside the INI program. For instance, the event `@ballDetection` is runnable if there is a ball detected by a video camera. RE is a part of $\mathbb{E}$. Inside the thread pool, several event instances may be evaluated at one time. Instructions of the action EA are evaluated in the same way as usual.

As mentioned earlier, $\mathbb{E}$ is the event instance generator, which has the characteristics as described below:

- $\mathbb{E}$ takes charge of registering events.
- $\mathbb{E}$ takes charge of generating an event instance when the (physical) given event occurs. It places the instance in RE (see the next point).
- $\mathbb{E}$ contains also a list of runnable event instances RE that correspond to events that physically occurred, but whose action is not yet under evaluation. In particular, event occurrences (or event callbacks) in RE are removed only after the event action EA has been pushed on the corresponding thread pool, or if the bound logical expression le is evaluated to `false` (see Section 4.2.2.10, page 108).
- $\mathbb{E}$ dialogues with INI operational semantics environment through: a) function calls, which register the events defined by the function b) event triggering c) event stopping and reconfiguring. (a) and (c) are actions of the environment on $\mathbb{E}$, while (b) is done in the reverse way.

There are three cases when invoking e_{inst}:

- The event of $e_{inst}.eid$ is terminated. In this case, it cannot be executed and it is removed from `RE`.

$$\frac{e_{inst} \in RE \quad T_{e_{inst}.eid} = true}{\mathbb{E}\ \langle cur,\ \Sigma\rangle \quad \Rightarrow \quad \mathbb{E}'\ \langle cur,\ \Sigma\rangle}$$

  in which, cur is either a current expression or rule being evaluated.

- At least one of the thread pools corresponding to the events on which e_{inst} is synchronized is not empty. In this case, e_{inst} cannot be executed. In other words, $\mathbb{E}$

needs to wait until all of the pools are empty (see the algorithm in Section 4.3, page 113).

$$\frac{e_{inst} \in RE \quad \exists k \in e_{inst}.L \quad P_k \neq \emptyset}{\mathbb{E}\ \langle cur,\ \Sigma\rangle \ \Rightarrow\ \mathbb{E}\ \langle cur,\ \Sigma\rangle}$$

- Otherwise, e_{inst} is executed.

$$\frac{e_{inst} \in RE \quad T_{e_{inst}.eid} = false \quad \forall k \in e_{inst}.L \quad P_k = \emptyset}{\langle cur,\ \Sigma\rangle \ \Rightarrow\ \langle cur,\ \Sigma'\rangle}$$

in which we have $\Sigma' = [\sigma'_v,\ \sigma'_f,\ [P'_1,\ T'_1, ..., P_{e_{inst}.eid} \cup Thread(e_{inst}.EA),\ T'_{eid}, ...]]$ and $Thread(e_{inst}.EA)$ is a new thread created for executing the event action.

4.2.2.9 Semantics of @init and @end

- The evaluation of a function begins with invoking the `@init` event (if exists), where below `@init` stands for its inside action:

$$\frac{\langle \texttt{@init},\ \Sigma\rangle \Rightarrow \langle \epsilon,\ \Sigma'\rangle}{\langle \triangleright,\ \Sigma\rangle \ \Rightarrow\ \langle [R, 0, 0],\ \Sigma'\rangle}$$

- When no more rules are applicable (which means that all the guards of the rules have been evaluated to `false` at least one time, in this case $j \geq r$) and no more event is running, we come to evaluate the `@end` event (if exists), where below `@end` stands for its inside action and "$\bullet$" indicates the "terminated" state of a function:

$$\frac{j \geq r \quad \forall k,\ T_k = true,\ P_k = \emptyset \quad \texttt{@end} \text{ exists}}{\langle [R,\ i,\ j],\ \Sigma\rangle \ \Rightarrow\ \langle \texttt{@end},\ \Sigma\rangle}$$

$$\frac{\langle \texttt{@end},\ \Sigma\rangle \Downarrow \langle \epsilon,\ \Sigma'\rangle}{\langle \texttt{@end},\ \Sigma\rangle \Downarrow \langle \bullet,\ \Sigma'\rangle}$$

- When no more rule is applicable, no more event is running, and the `@end` event does not exist, we finish evaluating the function:

$$\frac{j \geq r \quad \forall k,\ T_k = true,\ P_k = \emptyset \quad \texttt{@end} \text{ does not exist}}{\langle [R,\ i,\ j],\ \Sigma\rangle \ \Rightarrow\ \langle \bullet,\ \Sigma\rangle}$$

4.2.2.10 Semantics of Events when Used with Guards

For a shorthand, we use le instead of $e_{inst}.le$.

- An event is evaluated in the same way as in Section 7 (page 187) when the logical expression *le* is evaluated to `true`. Note that *le* is evaluated in the same context as with other independent rules.

$$\frac{e_{inst} \in RE \quad \langle le,\ \Sigma\rangle \Downarrow \langle true,\ \Sigma'\rangle}{\langle cur,\ \Sigma\rangle\ \Rightarrow\ \langle cur,\ \Sigma'\rangle}$$

  in which we have $\Sigma' = [\sigma'_v,\ \sigma'_f,\ [\{P'_1,\ T'_1\}, ..., \{P_{e_{inst}.eid} \cup Thread(e_{inst}.EA),\ T'_{eid}\}, ...]]$ and $Thread(e_{inst}.EA)$ is a new thread created for executing the corresponding event action.

- An event cannot be executed when the logical expression *le* is evaluated to `false`.

$$\frac{e_{inst} \in RE \quad \langle le,\ \Sigma\rangle \Downarrow \langle false,\ \Sigma'\rangle}{\langle cur,\ \Sigma\rangle\ \Rightarrow\ \langle cur,\ \Sigma'\rangle}$$

4.2.2.11 Semantics of Functions Operating on Events

This section involves a strong interaction between the execution contexts of $\overrightarrow{\mathcal{F}}$ and the event instance generator $\mathbb{E}$. For this reason, we make it explicit while still simplifying $\overrightarrow{\mathcal{F}}$ into a single execution context. The detailed explanation for the behavior of the two functions *register* and *unregister* is postponed to the next section (see Section 4.2.2.12, page 110).

Stopping Events Stopping an event with id *eid* means that the program does not allow any new instance of this event (the boolean variable T_{eid} is set to `true`). However, all existing threads in P_{eid} (those created before invoking the function `stop_event`) can still continue to run until they terminate.

$$\overline{\langle stop\_event([eid]),\ \Sigma\rangle\ \Rightarrow\ unregister(\mathbb{E}, l, \mathcal{C}_{eid}) \quad \langle \epsilon,\ \sigma'_v, \sigma_f, \Theta'_P\rangle}$$

in which in Θ'_P: $T_{eid} \leftarrow true$, and l is the location of the function evaluation context (in which the function `stop_event` is called) in $\overrightarrow{\mathcal{F}}$.

Reconfiguring Events Reconfiguring an event with id *eid* means changing the values of its input parameters. Normally, before reconfiguring an event, it should be stopped first. After reconfiguration, it can be restarted.

$$\frac{\begin{array}{c}\langle expr_1, \Sigma\rangle \Downarrow \langle iv_1, \Sigma'\rangle \\ \vdots \\ \langle expr_p, \Sigma^{(p-1)}\rangle \Downarrow \langle iv_p, \Sigma^{(p)}\rangle\end{array}}{\begin{array}{c}\langle reconfigure\_event(eid,\ [ip_1 = expr_1, ..., ip_p = expr_p]),\ \Sigma\rangle\ \Rightarrow \\ register(unregister(\mathbb{E}, l, \mathcal{C}_{eid}), l, \mathcal{C}'_{eid}) \quad \langle \epsilon,\ \Sigma^{(p)}\rangle\end{array}}$$

in which $\mathcal{C}'_{eid}$ differs from $\mathcal{C}_{eid}$ on: $ip'_1 \leftarrow iv_1, ip'_2 \leftarrow iv_2, ..., ip'_p \leftarrow iv_p$.

Restarting Events Restarting an event with id *eid* simply means that we allow it to be invokable again (i.e. the boolean variable T_{eid} is set back to `false`).

$$\frac{}{\langle restart\_event([eid]),\ \Sigma\rangle\ \Rightarrow\ register(\mathbb{E}, l, \mathcal{C}_{eid}) \quad \langle \epsilon,\ \sigma_v, \sigma_f, \Theta'_P\rangle}$$

in which in Θ'_P: $T'_{eid} \leftarrow false$.

4.2.2.12 Semantics of Evaluating Functions

The evaluation of a function call $f(pa_1, pa_2, ..., pa_m)$ involves the following steps:

1. Evaluate the arguments.
2. Create a new execution context for the callee Σ_{callee} and the associated proper execution context $\langle \Box, \Sigma_{callee}\rangle$, where $\Box$ is the place for the current instruction/rule (see Section 4.2.2.2, page 100).
3. Assign values to the callee's parameters (arguments) and initialize local variables $y_1, \cdots, y_p$ inside Σ_{callee}.
4. Evaluate the n input parameters $ip_1, ip_2, ..., ip_n$ of the events inside the callee's body.
5. Put the caller's context in "pause". However, the events of the caller still can run.
6. Register the events of the callee (events' callbacks are generated and notified to $\mathbb{E}$, and bound to the newly created execution context).
7. Evaluate events and rules in the callee (which corresponds to the proper evaluation of the body).

8. Wait until the callee stops.
9. Assign the returned value from the callee to the corresponding variable in the caller (except when the callee returns `void`).
10. Destroy the callee's execution context and remove the events of the callee from $\mathbb{E}$.
11. Continue to evaluate the caller.

We use the symbol $\triangleright$ to indicate the starting point of the function as mentioned earlier. $\triangleright$ is the `@init` event if it exists, or the first rule `[R, 0, 0]` if exists, or ϵ in case that there is none of them (see Sections 4.2.2.9 (page 108) and 7 (page 186)).

For the sake of readability, we introduce a metarule named `ParamEval`, which is used to evaluate parameters (both function's parameters and events' input parameters). In case for evaluating function's parameters, the inputs for that rule consist of:

- The event instance generator $\mathbb{E}$.
- The execution context Σ as mentioned in Section 4.2.2.2.
- The global execution context $\overrightarrow{\mathcal{F}}$.
- m expressions for m function's parameters: $pa_1, pa_2, \cdots, pa_m$.

The output for our rule is m values for m inputs, placed in the execution context associated to Σ along with the new $\mathbb{E}$, Σ, and $\overrightarrow{\mathcal{F}}$ after m steps as shown below:

$$ParamEval(\mathbb{E}, \Sigma, \overrightarrow{\mathcal{F}}, pa_1, pa_2, \cdots, pa_m) \Downarrow \mathbb{E}^{(m)}\ \overrightarrow{\mathcal{F}}^{(m)}\ ||\ \langle \{v_1, v_2, \cdots, v_m\},\ \Sigma^{(m)} \rangle$$

This rule can be unfolded as:

$$\begin{gathered}(\mathbb{E})\ \overrightarrow{\mathcal{F}}\ ||\ \langle pa_1,\ \Sigma \rangle\ \Downarrow\ (\mathbb{E}')\ \overrightarrow{\mathcal{F}}'\ ||\ \langle v_1,\ \Sigma' \rangle \\ \vdots \\ (\mathbb{E}^{(m-1)})\ \overrightarrow{\mathcal{F}}^{(m-1)}\ ||\ \langle pa_m,\ \Sigma^{(m-1)} \rangle\ \Downarrow\ (\mathbb{E}^{(m)})\ \overrightarrow{\mathcal{F}}^{(m)}\ ||\ \langle v_m,\ \Sigma^{(m)} \rangle\end{gathered}$$

where Σ' is equivalent to $\Sigma^{(1)}$.

Similarly, the `ParamEval` can be applied to evaluate `n` input parameters for all events:

$$ParamEval(\mathbb{E}, \Sigma, \overrightarrow{\mathcal{F}}, ip_1, ip_2, \cdots, ip_n) \Downarrow \mathbb{E}^{(n)}\ \overrightarrow{\mathcal{F}}^{(n)} \mid\mid \langle \{iv_1, iv_2, \cdots, iv_n\},\ \Sigma^{(n)} \rangle$$

The concrete semantics of function invocation in INI is defined as shown in the following rule:

$$\frac{\begin{array}{c} ParamEval(\mathbb{E}, \Sigma, \overrightarrow{\mathcal{F}}, \{pa_1, \cdots, pa_m\}) \Downarrow \mathbb{E}^{(m)}\ \overrightarrow{\mathcal{F}}^{(m)} \mid\mid \langle \{v_1, \cdots, v_m\},\ \Sigma^{(m)} \rangle \\ ParamEval(\mathbb{E}^{(m)}, [\rho_v + [y_1 \leftarrow \bot, \cdots, y_p \leftarrow \bot,\ x_1 \leftarrow v_1, \cdots,\ x_m \leftarrow v_m], \sigma_f, [\{\emptyset,\ false\}, ..., \\ \{\emptyset,\ false\}]], \overrightarrow{\mathcal{F}}^{(m)}, \{ip_1, \cdots, ip_n\}) \Downarrow \mathbb{E}^{(m+n)}\ \overrightarrow{\mathcal{F}}^{(m+n)} \mid\mid \langle \{iv_1, \cdots, iv_n\},\ \Sigma^{(n)}_{callee} \rangle \\ register(\mathbb{E}^{(m+n)},\ l,\ \overrightarrow{C}, \{iv_1, \cdots, iv_n\})\ \overrightarrow{\mathcal{F}}^{(m+n)} \mid\mid \langle wait,\ \Sigma^{(m+n)} \rangle \mid\mid \\ \langle \triangleright,\ \rho_v^{(n)},\ \sigma_f,\ [\{\emptyset, false\}, \cdots, \{\emptyset, false\}] \rangle_l \Downarrow \\ (\mathbb{E}^{(m+n+1)})\ \overrightarrow{\mathcal{F}}^{(m+n+1)} \mid\mid \langle wait,\ \Sigma^{(m+n+1)} \rangle \mid\mid \langle \bullet,\ \rho_v^{(n+1)},\ \sigma_f,\ [\{\emptyset, true\}, \cdots, \{\emptyset, true\}] \rangle_l \end{array}}{(\mathbb{E})\ \overrightarrow{\mathcal{F}} \mid\mid \langle f(pa_1, ..., pa_m),\ \Sigma \rangle\ \Downarrow\ unregister(\mathbb{E}^{(m+n+1)}, l, \overrightarrow{C})\ \overrightarrow{\mathcal{F}}^{(m+n+1)} \mid \langle \rho_v^{(n+1)}(f), \Sigma^{(m+n+1)} \rangle}$$

in which σ_f is the set of all function contexts (see more in page 186) $\sigma_f(f) = [\rho_v, R, \overrightarrow{C}]$, l is the location of the function evaluation context in $\overrightarrow{\mathcal{F}}$, and $\bullet$ is used to indicate the end state of a function. Σ_{callee} is the following execution context for the callee: $[\rho_v, \sigma_f, [\{\emptyset,\ false\}, ..., \{\emptyset,\ false\}]]$, where ρ_v (the variable and parameter context for the callee) has been drawn from $\sigma_f(f)$, and there are as many empty thread pools (i.e. they are not yet active, since the events still need to be registered) as callbacks $\overrightarrow{\mathcal{C}}$ in $\sigma_f(f)$.

The first three lines in the premise part indicates the evaluation for the callee's parameters (line 1) and its local variables (lines 2-3). The next three lines shows the process of registering events , binding values to parameters and local variables, and evaluating all events and rules inside the callee. The conclusion part just says that after we evaluate the callee, we implicitly unregister its events and get its returned value. Indeed, the (meta-) function *register* yields a new event instance generator, where the callbacks $\overrightarrow{\mathcal{C}}$ have been bound to the execution context $\mathcal{F}_l$ while the (meta-) function *unregister* performs the reverse operation.

To illustrate the general idea, let us consider a simple case, in which a function has one event (with id eid_1 and has one input parameter ip_1), one parameter pa_1 ($m = 1$), and one local variable y_1 ($n = 1$). Also for simplicity, we assume that $\overrightarrow{\mathcal{F}}$ is empty:

$$\frac{\begin{array}{c}(\mathbb{E})\ \overrightarrow{\mathcal{F}}\ ||\ \langle pa_1,\ \Sigma\rangle \Downarrow \mathbb{E}^{(1)}\ \mathcal{F}^{(1)}\ ||\ \langle v_1,\ \Sigma^{(1)}\rangle \\ (\mathbb{E}^{(1)})\ \overrightarrow{\mathcal{F}}^{(1)}\ ||\ \langle ip_1,\ [\rho_v + [y_1 \leftarrow \perp,\ x_1 \leftarrow v_1], \sigma_f, [\{\emptyset,\ false\}]]\rangle \Downarrow \mathbb{E}^{(2)}\ \mathcal{F}^{(2)}\ ||\ \langle iv_1,\ \Sigma^{(1)}_{callee}\rangle \\ register(\mathbb{E}^{(2)}, l, C_{eid_1}, \{iv_1\})\ \overrightarrow{\mathcal{F}}^{(2)}\ ||\ \langle wait,\ \Sigma^{(2)}\rangle\ || \\ \langle \triangleright, \rho'_v,\ \sigma_f,\ [\{\emptyset, false\}]\rangle_l \Downarrow \\ (\mathbb{E}^{(3)})\ \overrightarrow{\mathcal{F}}^{(3)}\ ||\ \langle wait,\ \Sigma^{(3)}\rangle\ ||\ \langle \bullet,\ \rho^{(2)}_v,\ \sigma_f,\ [\{\emptyset, true\}]\rangle_l\end{array}}{(\mathbb{E})\ \overrightarrow{\mathcal{F}}\ ||\ \langle f(pa_1),\ \Sigma\rangle\ \Downarrow\ unregister(\mathbb{E}^{(3)}, l, \overrightarrow{C})\ \overrightarrow{\mathcal{F}}^{(3)}\ |\ \langle \rho^{(2)}_v(f),\ \Sigma^{(3)}\rangle}$$

4.3 Event Synchronization

In this part, we introduce the detailed algorithm that was applied for synchronization in INI, that is a part of $\mathbb{E}$ and implemented in Java within the INI evaluator (adapted from [LHMP11]).

Algorithm 1 Our algorithm to execute e_0 synchronized with $(e_1, e_2, ...e_N)$

```
1:  while ¬allLocked do
2:    allLocked := true
3:    lock(l_0)
4:    for i := 1 to N step 1 do
5:      if ¬ tryLock(l_i) then
6:        allLocked := false
7:        for j := 1 to i − 1 step 1 do
8:          unlock(l_j)
9:        end for
10:       unlock(l_0)
11:       sleep(randomTime())
12:       break
13:     end if
14:   end for
15: end while
16: for i := 1 to N step 1 do
17:   wait-until count_i = 0
18: end for
19: count_0 := count_0 + 1
20: for i := 1 to N step 1 do
21:   unlock(l_i)
22: end for
23: unlock(l_0)
24: eval(e_0)
25: count_0 := count_0 − 1
```

To implement synchronization in INI, we use one lock and one *count* variable associated

with each event. The lock is an instance of `java.util.concurrent.locks.ReentrantLock` and is used to avoid concurrent execution when required. We use the functions `lock` (blocking), `tryLock` (non-blocking and returning `true` of `false` depending on whether the locking was successful or not), and `unlock`. For more details, please refer to the Java documentation for the methods of the same names in the `ReentrantLock` class [Ora13a]. The *count* variable holds the number of threads currently executing for the associated event. We use the following notation: for an event e_i, we call l_i its associated lock and $count_i$ its associated thread counting variable.

Let $(e_1, e_2, ..., e_N)$ be the list of target event ids with which e_0 synchronizes. To execute the event bound to e_0 in INI, we apply Algorithm 1, which also applies to any event execution in the INI system. We can see that the execution of events includes four steps. First, e_0 locks its own lock and tries to lock all target events (lines 1-15). When all events are locked, the event e_0 needs to wait until all other event instances are terminated (lines 16-18). The `while-do` mechanism in our algorithm is currently implemented with Java monitors and thread notification. Next, the number of threads executing for event e_0 is incremented and locks for all events are released (lines 19-23). Finally, the event can be actually evaluated and when it is terminated, the number of running threads for it is decremented (lines 24-25).

4.4 Summary

In this chapter, we described how programmers may write INI programs. Events and rules may be defined independently or in combination. Other features like lists, regular expressions, set selection expressions, binding to Java objects, etc. are also fully mentioned. Additionally, to help programmers understand more about their code, we showed operational semantics for our language that brings out unambiguously the meaning of how INI works, particularly the manner of which events and rules are managed and triggered.

In our work, we omit the semantics rule for evaluating $\langle n \text{ of } ALGEBRAIC\_TYPE, \Sigma \rangle$. Formalizing such a rule would force us to extend Σ, by adding some "stores" in order to record all the instances of any algebraic data type every time we meet an assignment (see

more in Section 5.2.1.2, page 119). INI has its own mechanism to map instances to their corresponding types. Reflecting this feature in the semantics would complicate it even more for a very limited benefit. This is the only construction that is not in the scope of the current operational semantics.

Chapter 5

Static Analysis for INI Programs

In this chapter, we present our work on ensuring quality of INI programs. First, we give an overview of static analysis technique that is one of the hot topics in software engineering (Section 5.1). Next, we discuss the type system of INI, including type inference and type checking engines to avoid type conflicts (Section 5.2). Subsequently, we show how INI can be converted to Promela, the input modeling language of the model checker SPIN (Section 5.3). Then SPIN can help us to verify constraints or properties that need to be satisfied.

5.1 Introduction to Static Analysis

Static analysis, one kind of quality assurance activities, is "the detection of real or potential problems by analyzing the source code without its execution (proscriptive analysis) and/or providing explanations about program behavior (descriptive analysis)" [LS09]. Indeed, this is a powerful method for the detection of possible anomalies. We use the term "anomaly" instead of defect since it may or may not cause the program to fail. However, those anomalies may help programmers understand more about their programs and give hints when errors occur [Hua09].

Some techniques employed in static analysis are type checking, logical statement checking, interface and include problem checking, source code crawler, program transformation and refactoring, source level software metrics, bad smell detection, model checking, etc [LVS09]. To understand more about static analysis, please refer to [SWH12, LS09, Hua09, Bou13]. In the next sections, we introduce how we apply static analysis for making INI

programs more dependable, through a type system and a tool to convert INI to Promela, the input modeling language of the model checker SPIN.

5.2 Type System of INI

5.2.1 Overview

Informally, a type system consists of (1) a mechanism to define types and associate them with certain language constructs, and (2) a set of rules for type equivalence, type compatibility, and type inference [Sco09]. Based on the characteristics of its type system, a language can be classified as:

- Strongly typed (e.g. Ada, Java, Haskell, Python, etc.): variables must be strictly used in a consistent way with their types.
- Weakly typed (e.g. Perl, C/C++, etc.): several manners are allowed to bypass the type system.
- Statically typed (e.g. Java, Haskell, Miranda, etc.): type checking happens at compile time.
- Dynamically typed (e.g. Python, Ruby, etc.): type checking happens at run time.

A language may have a mix, e.g. Java has a mostly static type system in conjunction with some runtime checks.

A strong type system helps the compiler to detect and avoid ill-formed programs which lead to runtime errors. To understand more about types in programming languages, please refer to [Har00, Pie04, DL10, Pie02].

Since context-ware applications are widely used in many important domains, it is necessary that all those programs are well-typed. For this reason, we develop INI with a strong and static type system. In the following parts, we detail the type system in INI, involving supported data types, type inference and type checking engines.

5.2.1.1 Built-in Types

INI comes with five built-in numeric types for numbers (*Double*, *Float*, *Long*, *Int* and *Byte*), a *Char* type, and a *Boolean* type. Besides, INI provides a built-in polymorphic map type: $Map(K, V)$, where K is the key type and V is the value type. Lists are instances of maps when $K = Int$ (in reality, it is more of an indexed set). *String* type is a list of *Char*. Syntactically, lists can be noted with the * notation: $T^* \mathrel{\widehat{=}} Map(Int, T)$.

INI types are ordered with a subtyping relation $\succ$. Numerics are ordered so that it is impossible to assign more generic numbers to less generic numbers: $Double \succ Float \succ Long \succ Int \succ Byte$. Other conversions among numbers must be done by using built-in functions such as `to_byte`, `to_int`, `to_long`, `to_float`, and `to_double` [LP12].

5.2.1.2 Algebraic Data Types

Product Types To define a new type, programmers use the *type* keyword followed by a name starting by an uppercase letter. For example, we can define and use a *Person* type as:

```
type Person = [name:String, age:Int]
p = Person[name="Giang", age=29]
println("Information:␣" + p.name + "␣is␣" + p.age)
```

Field access is done with a usual "dot" and field initialization is not mandatory. One can construct a person with undefined age or name.

Sum Types (a.k.a. Union Types or Compound Types) Sum types are an extension of simple structured types that allow the definition of types that have different constructors. In INI, they are defined and used in a very similar way and can be related to Abstract Syntax Tree (AST) in the sense that they allow the definition of typed trees structures.

For instance, we can define a recursive type for arithmetical expressions that include typical operations on floating point numbers.

```
type Expr = Number[value:Float]
          | Plus[left:Expr,right:Expr]
          | Mult[left:Expr,right:Expr]
          | Div[left:Expr,right:Expr]
          | Minus[left:Expr,right:Expr]
```

```
          | UMinus[operand:Expr]
```

Once constructed, programmers can define any expression using the constructors. For example, to define the expression `-(3.0*2.0+1.0)`:

```
expr = UMinus[operand=Plus[
    left=Mult[
        left=Number[value=3.0],
        right=Number[value=2.0]],
    right=Number[value=1.0]]]
```

5.2.1.3 Type Set Selection Expressions

In Section 4.1.7.4 (page 93), we have seen the set selection expression that allows to select values within a set. Each object constructed with a user type constructor is automatically part of an instance set, which is named after the constructor name. Thus, it is possible to select instances using the set selection construct. For example, the following rule raises an error if a number has an undefined value:

```
n of Number | !n.value {
    error("Invalid␣number␣value")
}
```

Note that sets are local to the functions that construct the instances.

5.2.1.4 Pattern Matching

As shown earlier, INI enables programmers to construct algebraic types, which normally are used to define complex structures. In order to dig into these structures, INI supports a match operator, which is similar to the match construct in advanced functional programming languages such as Caml [1] [CM98] and Haskell [Tho96]. To demonstrate the use of such types, we show an INI program (Listing 5.2.1) that allows the construction of algebraic expressions, and their evaluation with the `calc` function. This function uses the match operator within the rules guards to switch to the right action depending on the actual type constructor.

Listing 5.2.1 *Using algebraic types with pattern matching in INI.*

[1] Caml is an acronym for "Categorical Abstract Machine Language".

```
type Expr = Number[value:Float]
          | Plus[left:Expr,right:Expr]
          | Mult[left:Expr,right:Expr]
          | Div[left:Expr,right:Expr]
          | Minus[left:Expr,right:Expr]
          | UMinus[operand:Expr]

function main() {
    @init() {
        expr = UMinus[operand=Plus[
            left=Mult[
                left=Number[value=3.0],
                right=Number[value=2.0]],
            right=Number[value=1.0]
        ]]
    }
    //This rule can be added to check whether the operators
    //were correctly constructed
    op of Plus | (!op.left) || (!op.right) {
        error("invalid␣plus␣operator")
    }
    @end() {
        println("The␣value␣of␣"+expr_string(expr)+
         "␣is␣"+expr_value(expr))
    }
}
//Calculates the expression.
function expr_value(expr) {
    expr ~ Number[] {
        return expr.value
    }
    expr ~ Plus[] {
        return expr_value(expr.left)+expr_value(expr.right)
    }
    expr ~ Mult[] {
        return expr_value(expr.left)*expr_value(expr.right)
    }
    expr ~ Minus[] {
        return expr_value(expr.left)-expr_value(expr.right)
    }
    expr ~ Div[] {
        return expr_value(expr.left)/expr_value(expr.right)
    }
    expr ~ UMinus[] {
        return -expr_value(expr.operand)
    }
}
```

Note that in the guard, it is not mandatory to declare the whole structure of the type for matching. For example, `expr` ∼ `Plus[]` and `expr` ∼ `Plus[left,right]` are equivalent and

give the same results.

5.2.1.5 Function Types

A function type is noted as `(T1,T2,...,Tn)->T` and is associated to any function definition, where `Ti` is the expected type of the i^{th} parameter, and `T` is the function's return type.

5.2.1.6 Polymorphism

INI supports polymorphic functions, which are those functions that have arguments of "variable" types. For example, let us consider an INI program counting occurrences in a list as shown in Listing 5.2.2.

Listing 5.2.2 *Counting occurrences in a list.*

```
/*
 * This example shows how to count element occurrences
 * in a list.
 */
function main() {
    @init() {
        //In INI, strings are lists of characters
        s = "This␣is␣the␣string␣we␣will␣count"
        println("Counting␣'"+s+"'")
        c = countOccurrencesAndStoreToMap(s)
        println("Number␣of␣e(s):␣"+c['e'])
        println("Number␣of␣a(s):␣"+c['a'])
        println("Number␣of␣s(s):␣"+c['s'])
        println("Number␣of␣i(s):␣"+c['i'])
        println("Number␣of␣spaces:␣"+c['␣'])
        l = [1, 2, 1, 7]
        println("Counting␣'"+l+"'")
        c2 = countOccurrencesAndStoreToMap(l)
        println("Number␣of␣1:␣"+c2[1])
        println("Number␣of␣7:␣"+c2[7])
        println("Number␣of␣3:␣"+c2[3])
    }
}

/*
 * The results for each element is stored in a map
 * (each element being a key).
 */
function countOccurrencesAndStoreToMap(s) {
    @init() {
```

```
31          i=0
32      }
33      i < size(s) {
34          c[s[i++]]++
35      }
36      @end() {
37          return c
38      }
39  }
```

The `countOccurrencesAndStoreToMap` function is polymorphic. In our case, the inferred functional type is `(Map(T,Int))->Void`, where `T` is a type parameter so that both its invocations in the function `main` are correct. Actually, the INI type inference algorithm finds the most general type so that functions are only typed with the least possible constraints and remain polymorphic when possible.

5.2.2 Type Inference in INI

Here is a rule which states that undefined values have any type, i.e. their type is an unconstrained type variable.

$$\text{(NULL)}\ \frac{}{\vdash null : T}$$

Generally, type inference is the process of determining the types of names and expressions based on the known types of some symbols that appear in them [Mit03, CT11]. Type inference may be viewed as a separate operation performed during type checking, or it may be considered to be a part of type checking itself [LL11].

INI provides type inference, so that programmers may leave types implicit. For instance, the `i=0` statement will assign to `i` the `Int` type. If programmers tries to use i with another type within the definition scope of `i`, (for instance with the `i=0.1` statement that assigns to `i` a `Float` type) the INI type checker will raise a type mismatch error. Following the same type inference principles, accessing a variable with the square brackets map access construct will define the type of the variable to be a map. For instance, the `l[i]` expression automatically tells INI that `l` is of type `Map(Int,T)` (i.e. a list of `T`), where `T` can be any type.

Most types in INI are calculated with the type inference engine. The core of this type inference is based on a Herbrand unification algorithm, as depicted by Robinson [Rob65, RV01, Har09, Mob09]. The typing algorithm is enhanced with polymorphic function support, abstract data types (or algebraic types) support, and with internal sub-typing for numeric types. Type constraints are gathered with the rules depicted in the next parts.

5.2.2.1 The Match Operator

Within a guard a match operator can be used to match strings with regular expressions and bind the results to some variables. For instance, `"a b c" ~ regexp("(.) (b) (.)",v1,v2,v3)` will match and be evaluated to `true`. Since there are three groups (between parenthesis), the three match sub-results will be bound to the given variables. Thus, once the match is done, we will have `v1="a"`, `v2="b"`, and `v3="c"`. With regard to typing, all the bound variables are strings (see Sections 4.1.8 (page 94) and 4.2.2.4 (page 103) to understand more about this operator).

5.2.2.2 Map Access

A map access node such as `x[y]` allows to say that the resulting type of the expression is `V`, if `x` is of type `Map(K,V)` and `y` is of type `K`.

$$\text{(MAPACCESS)}\ \frac{\vdash x : Map(K,V) \qquad \vdash y : K}{\vdash x[y] : V}$$

5.2.2.3 Field Access

For a field access such as `x.f`, the type of the accessed expression `x` must have a field of the right name and type. Note that, when using match expressions, by construction, it is often the case that the type of the accessed expression is already resolved.

$$\text{(FIELDACCESS)}\ \frac{\vdash T.f : V}{\vdash (x : T).f : V}$$

5.2.2.4 Function Invocation

For an invocation, the rule implies some constraints between the function type and the type of the result and of the parameter expressions. To support polymorphism, the function's body is evaluated within its own environment, so that each invocation has its own set of constraints.

$$\text{(INVOCATION)}\ \frac{\vdash f : T_1, T_2, ..., T_n \to T \qquad \{\vdash e_i : T_i\}_{i\in[1..n]}}{\vdash f(e_1, e_2, ..., e_n) : T_1, T_2, ..., T_n \to T}\ [new(env)]$$

5.2.2.5 List Definition

List definitions are done with the following syntax: `[e1, e2,...,en]`, where all the expressions must be of the same type `T`. Then the type of the resulting list is T^* (equivalent to `Map(Int,T)`).

$$\text{(LISTDEF)}\ \frac{\{\vdash e_i : T\}_{i\in[1..n]}}{\vdash [e_1, e_2, ..., e_n] : T^*}$$

5.2.2.6 Return Statement

A `return` statement implies that the enclosing function's return type is `Void`. Note that, in addition, the function's return type is `Void` if no `return` statements appear in the function's body.

$$\text{(RETURN)}\ \frac{\vdash f : \_ \to Void}{\vdash (\texttt{return}) \in f}$$

A `return` statement with an expression implies that the enclosing function's return type is of the expression's type.

$$\text{(RETURNEXPR)}\ \frac{\vdash f : \_ \to T \qquad \vdash e : T}{\vdash (\texttt{return}\ \ e) \in f}$$

5.2.2.7 Type Instantiation

One can construct an instance of an algebraic type using one of its constructor. For instance, a algebraic type defined as: type `A = C[f1:T1, f2:T2,...,fn:Tn]` has one constructor `C` with `n` typed fields. A constructor `C` of an algebraic type `A` is typed with

a type of the same name (`C`), along with the subtyping relation $C \preceq A$ (see more in Section 5.2.2.8). When an instance of `A` is constructed using `C` through the expression `C[f1=e1,f2=e2,...,fn=en]`, then the type of each expression `ei` must be of the type of the field `fi`, as declared in the algebraic type definition (i.e. `Ti`).

$$\text{(CONSTRUCTOR)}\ \frac{\{\vdash C.f_i : T_i\}_{i\in[1..n]} \qquad \{\vdash e_i : T_i\}_{i\in[1..n]}}{\vdash \texttt{C[}f_1\texttt{=}e_1\texttt{,}\, f_2\texttt{=}e_2\texttt{,}\, ...\texttt{,}\, f_n\texttt{=}e_n\texttt{]} : C}$$

where `T` is the type that corresponds to the `C` constructor, which is a part of an algebraic data type.

5.2.2.8 Subtyping Rules

INI support subtyping, which means that if `A` is a subtype of `B`, any term of type `A` can be safely used in the context where a term of type `B` is expected.

$$\frac{\vdash x : A \qquad A \preceq B}{\vdash x : B}$$

5.2.2.9 Integer Set Declaration

An integer set declaration allows the programmers to declare an integer domain with min and max bounds. The syntax is `[e1..e2]` where `e1` is the min bound expression and `e2` is the max one. Both min and max bound expressions must be of type `Int`, and the resulting type is a `Set(Int)`.

$$\text{(INTSET)}\ \frac{\vdash e_1 : Int \qquad \vdash e_2 : Int}{\vdash [e_1\,..\,e_2] : Set(Int)}$$

5.2.2.10 Set Selection

Within a set expression, one can select elements in a set, and bind them to variables. In that construct, each variable is of the type of the set elements.

$$\text{(SELECTION1)}\ \frac{\{\vdash x_i : T\}_{i\in[1..n]} \qquad \vdash s : Set(T)}{\vdash x_1, x_2, ..., x_n\ \texttt{of}\ s}$$

A selection operation can also be done within a constructor type `C`, i.e. it will select instances that have been constructed through this constructor.

$$\text{(SELECTION2)}\ \frac{\{\vdash x_i : C\}_{i\in[1..n]}}{\vdash x_1, x_2, ..., x_n\ \texttt{of}\ (\texttt{C} : Set(C))}$$

5.2.3 Type Checking in INI

Type checking is the activity of ensuring that the operands and the operators are of compatible types. A compatible type is one that either is legal for the operator or is allowed under language rules to be implicitly converted by compiler-generated code (or the interpreter) to a legal type [Seb12]. Type checking is one kind of semantics analysis [Med08].

INI performs type checking at compile time (statically typed). In details, to avoid type conflicts (clashes), INI works with the following steps:

1. The parser constructs the AST.
2. An AST walker constructs the typing rules that should be fulfilled for each AST node and adds them to a constraint list.
3. A unification algorithm is run on the type constraints. If conflicts are detected, they are added to the error list.
4. If errors are found, they are reported to the programmers and INI does not proceed to the execution phase. Otherwise, the program will be evaluated normally.

5.3 Model Checking INI Programs

5.3.1 Introduction to Model Checking

IEEE defines Verification and Validation (V&V) as "the process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements" [IEE91].

In software engineering, software V&V is defined to be "an engineering practice which provides confidence that the system software was built adequately and will meet the needs of the system" [Fis06]. V&V techniques can be applied at different phases during the

software development life-cycle that correspond to design quantification, installation quantification, operational quantification and performance quantification. Software V&V, along with planning and configuration management are the key components to guarantee the success of a software project [Boa95a, Fai09]. To learn more about software V&V, please see [Boa95b, LS09, Eng10].

Among V&V methods and activities, model checking emerges as an efficient technique for automatic verification of software and reactive systems [BBF+10].

> *"Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model [BK08]".*

The key idea is that when a faithful model of a system is given, tools can verify automatically whether the system fulfills the requirements or not. The verification process is an exhaustive search of the state space of the design for a counterexample. If a counterexample is found, the system does not meet the requirements and we get hints on where the requirement fails (see more in [CGP99, Kat99, Mer01]). State-of-the-art model checkers can handle state spaces of about 10^8 to 10^9 states with explicit state-space enumeration. Using clever algorithms and tailored data structures, larger state spaces (10^{20} up to even 10^{476} states) can be handled for several specific problems [BK08].

Basically, applying model checking techniques involves the following phases as shown in Figure 5.1:

1. Modeling phase:
   - Construct a model for the system (manually or automatically) by using the modeling language dedicated to each model checker.
   - Formalize the property (desired behavior) needed to be checked by using the property expression formulas or a dedicated language.
2. Running phase: run the model checker on the inputs (i.e. the model and the formalized property) to check whether the model fulfills the property or not.

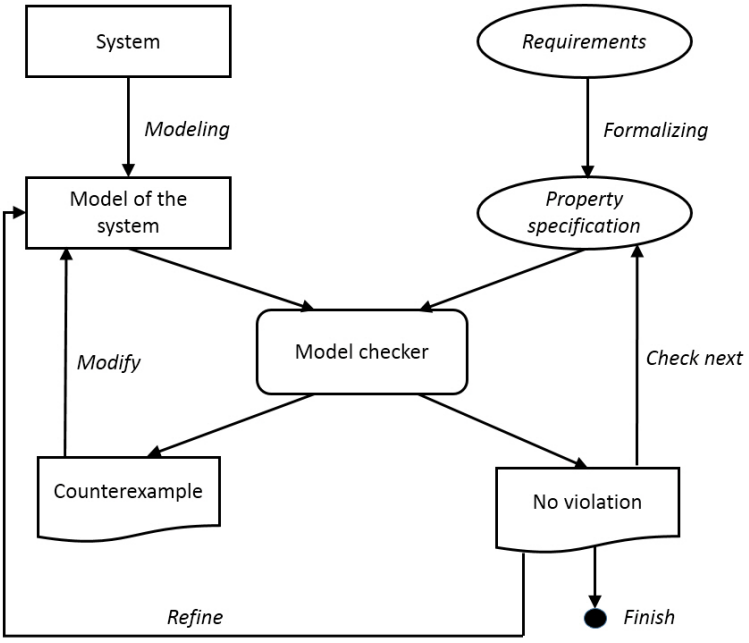


Figure 5.1: The methodology of model checking [Kat99].

3. Analysis phase:

   - If the property is satisfied, users may continue to check other properties.
   - If the property is violated, users may dig into the result given by the model checker and:
     - Use the simulation feature to repeat all steps leading to the counterexample.
     - Evaluate whether the counterexample is realistic or not.
     - Refine the model or property if needed, or modify the system.
   - If there is a problem related to out-of-memory, users need to make the model less complex to reduce the number of states and then check again.

Applying model checking has several advantages:

- Various possible applications: hardware/software verification, communication protocols, embedded systems, etc.
- Support of partial verification: may verify only important properties and temporarily ignore the insignificant ones.

- No need for proofs and the users directly see counterexamples.
- Saves time and efforts when compared with exhaustive testing or simulation.
- Can be applied at early phases during the software development life-cycle.
- Has sound and strong mathematical foundations.
- Attracts more and more interest and supports from community (in both academy and industry).

However, the primary limitation of model checking is that if there are too many processes and data paths in the program, it leads to state explosion. Several current research topics are devoted to solve this problem.

In the next parts, we briefly introduce the model checker named SPIN and its modeling language called Promela. The reason we select SPIN is that this tool is appropriate for verifying concurrent and reactive systems [Hol03, BK08]. Then we present our developed tool called INICheck, which can convert a significant subset of INI to Promela. Several examples also will be shown to illustrate the possible applications of using this tool.

5.3.2 Model Checking with SPIN

5.3.2.1 Overview

SPIN [2] [SPI13] is "one of the leading model checking tools used by professional software engineers" [BA10]. Figure 5.2 (adapted from [Hol97]) shows the structure of simulation and verification in SPIN. The inputs of SPIN are the model written in Promela and specifications expressed as assertions, labels, never claims, and Linear Temporal Logic (LTL) formulas. The SPIN tool generates a customized explicit-state model checker and runs the program for verification. SPIN is an on-the-fly model checker, which means that whenever it meets a counterexample to the specifications, it stops and shows the errors. Users can moreover reconstruct all steps leading to the error. SPIN takes advantage of many state-of-the-art techniques to speed up the model checking process and save memory such

[2] SPIN stands for "Simple Promela Interpreter".

as partial order reduction, state compression, bitstate hashing, and weak fairness enforcement [BA08, Hol03].

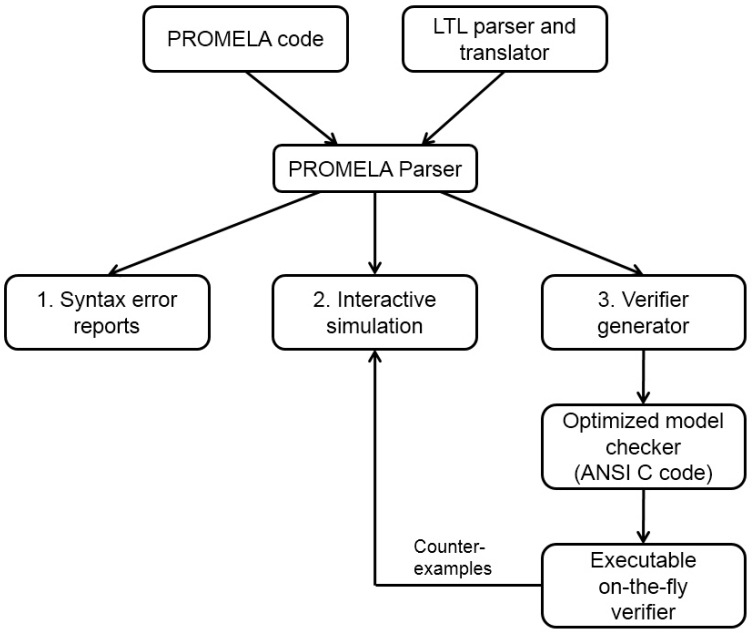


Figure 5.2: Overview of SPIN [Hol97].

Several researchers already tried to convert classical programming languages to Promela for model checking with SPIN. Some achievements have already been obtained even though not comprehensive. For example, Havelund *et al.* [Hav99] presented Java PathFinder (JPF), a tool for translating Java to Promela. They translated a non-trivial subset of Java (e.g. class, method, statement, variable) to Promela. JPF allows a programmer to annotate his or her Java programs with assertions and verify them using the model checker SPIN. One of their limitations is that they need to perform the abstractions by hand first, and then translate the simplified Java program to Promela using JPF. Jiang *et al.* [Ke09] translated a subset of C to Promela. One of the largest limitations of their work is that they did not fully support pointers, which is the core concept of C. Since Promela supports embedded C code, some authors handled "untranslatable" C code by embedding it directly. Faria *et al.* [FMSP11] provided mechanisms to extract a model from an Ada program to be used with SPIN. The extracted model covers a subset of Ada features and closely relates to the corresponding Ada program.

5.3.2.2 Promela

Promela [3] is an imperative language inspired from C, with constructs to handle concurrent processes in a system. The description of a concurrent system includes one or several process definitions and at least one of them must be active. Communication between processes may be carried out via synchronous/asynchronous channels. Promela statements are either executable or blocked, depending on the related condition.

Generally, a Promela model consists of:

- Type declarations (optional).
- Channel declarations (optional).
- Variable declarations (optional).
- Process declarations.
- The `init` process (optional).

Proctypes A Promela program consists of processes, each process is declared by one `proctype` and at least one `proctype` should be active (by declaring with the keyword `active` or by invoking the `run` command). Processes have the following characteristics:

- They execute concurrently (interleaving) with all other processes.
- They communicate with each other process via channels (as shown later).
- They can access shared variables.

Each process contains: a process identifier, a parameter list, and a sequence of variable declarations and statements. The `init` keyword is used to declare the behavior of a process that is active in the initial system state. The `init` process has no parameters.

We can see that processes have several commonalities with functions and events in INI, such as having parameters and their concurrent invocation and execution. We take into account this remark when converting INI to Promela.

[3] Promela stands for "Process or Protocol Meta Language".

Data Types Promela provides some familiar basic data types [BA08]: `bit`, `bool`, `byte`, `short`, `int`, `unsigned`. It also supports arrays, records (as structs in C). However, in Promela:

- There is no separate character type.
- There is no string variable.
- There is no floating-point data type.

All variables are initialized by default to zero. In Promela, the keyword `mtype` can be used for defining symbolic values. Besides, like in INI, programmers may define their own types. For example, programmers may define a new type called `MESSAGE` as follows:

```
typedef MESSAGE {
    mtype message;
    byte source;
    byte destination;
    bool urgent
}
```

Anonymous variables are denoted by an underscore (_). In Promela, type conflicts are detected at runtime.

Operators and Expressions Promela comes with a basic support for simple operators and expressions working on numeric and boolean values, such as:

- Addition, subtraction, multiplication, division, modulo: $+$, $-$, $*$, $/$, $\%$.
- Increment, decrement: $++$, $--$.
- Comparison operators: $>$, $<$, $>=$, $<=$, $==$, $!=$.
- Logical operators: $\&\&$, $||$.
- Bitwise operators: $\&$, $\hat{}$, $|$.

Statements Statements are separated by a semicolon (`;`) or, equivalently, an arrow (`->`). Note that semicolons are separators, not statement terminators. A statement in Promela has two states: executable (i.e. the statement can be executed immediately) or blocked (i.e. the statement cannot be executed). For example:

- $9 > 1$: always executable
- $x < 9$: only executable if `x` is smaller than 9.
- $!x$: only executable if `x` is `false` or 0.
- $(x + 5) -> ...$: only executable if `x` is not equal to -5.

Selection Statements Promela supports the `if`-statement as in other classical languages. The syntax is shown below:

```
if
    ::(guard_expression_1) -> statement_list_1;
    ::(guard_expression_2) -> statement_list_2;
    ::(guard_expression_3) -> statement_list_3;
    ...
fi
```

However, in Promela, there is no meaning to the order of the alternatives, which differs fundamentally from other languages. If more than one guard statement is executable, one of them will be selected non-deterministically. To illustrate the `if`-statement, let us show a simple piece of Promela code (as shown in Listing 5.3.1) which finds the max of two numbers `x` and y.

Listing 5.3.1 *Finding the larger one among two numbers in Promela.*

```
1 if
2     :: x>=y -> m  = x
3     :: x<=y -> m = y
4 fi
```

Repetitive Statements The `do`-statement in Promela bears similarities to the `if`-statement, in that it includes the evaluations of the guards, followed by a list of statements to express the desired action. However, one different feature is that after the completion of

a sequence of statements, the execution comes back to the beginning of the `do`-statement, i.e. we start a new cycle. When many guards are evaluated to `true`, the choice of executing a branch is nondeterministic. The syntax for the `do`-statement is shown below:

```
do
    ::(guard_expression_1) -> statement_list_1;
    ::(guard_expression_2) -> statement_list_2;
    ::(guard_expression_3) -> statement_list_3;
    ...
od
```

We can see that the `do`-statement has something in common with rules in INI, namely that when at least one guard condition is true, the associated action is evaluated and the loop goes on. The loop stops only when all the guards are evaluated to `false`.

To illustrate the `do`-statement, let us take a simple traffic light control system as shown in Listing 5.3.2 (taken from [BA08]):

Listing 5.3.2 *A simple traffic light control system.*

```
1 mtype= { red, yellow, green };
2 mtype= { green_and_yellow, yellow_and_red };
3 mtype light = green;
4
5 active proctype P() {
6     do
7         ::if
8             :: light == red -> light = yellow_and_red
9             :: light == yellow_and_red -> light = green
10            :: light == green -> light = green_and_yellow
11            :: light == green_and_yellow -> light = red
12        fi;
13        printf("The␣light␣is␣now␣%e\n", light)
14    od
15 }
```

Concurrency Promela allows several processes running in parallel in terms of interleaving. One solution to avoid undesirable interleaving problems is by using atomic sequences as explained later.

Channels In Promela, besides shared variables, communication among active processes can also be done via message channels, which are appropriate for modeling notification

mechanisms or protocols. Channels are declared using the keyword `chan` as follows:

```
chan <name> =[<capacity>] of {<t1>,<t2>,...,<tN>}
```

in which, `capacity` is the number of elements that can be buffered in the channel and `<t1>,<t2>,...,<tN>` are the types of the elements that will be transmitted over the channel.

For sending and receiving messages through channels, programmers may use the following statements:

```
//Sending message
channel_name ! send_args
//Receiving message
channel_name ? recv_args
```

If `capacity` is 0, the channel has zero buffering size, which means that it can pass but not store the messages. Additionally, in this case, the synchronous handshake (rendezvous) communication will be set up. For instance, if we have two processes `A` and `B`, along with one declared channel:

```
chan ctest = [0] of {byte}
```

Let us consider a scenario that process `A` wants to send 5 to `ctest` by invoking the statement as shown below:

```
1 proctype A {
2     ...
3     ctest ! 5
4     ...
5 }
```

Process A will be blocked until another process tries to receive 5. For instance, if there is a receive statement in process `B`:

```
1 proctype B {
2     ...
3     ctest ? msg
4     ...
5 }
```

then process `A` will be unlocked and the variable `msg` in process `B` will be 5.

A classical example illustrating communication through channels in Promela is a client-server model, in which the server replies to requests from the clients, as displayed in Listing 5.3.3 (taken from [BA08]):

Listing 5.3.3 *A client-server model in Promela.*

```
chan request = [0] of {byte};
chan reply = [0] of {bool};
active proctype Server() {
    byte client;
    end:
    do
        :: request ? client ->
            printf("Client␣%d\n", client);
            reply ! true
    od
}

active proctype Client0() {
    request ! 0;
    reply ? _
}
active proctype Client1() {
    request ! 1;
    reply ? _
}
```

In other circumstances, when the buffering capacity of a channel is set as larger than zero, messages are stored in first-in first-out order. For example, the following channel can store up to 16 messages:

```
chan a = [16] of { short }
```

Besides, the communication among processes in that case is asynchronous, which means that the sender process does not need to wait the receiver process.

Atomic Sequences of Statements If a sequence of statements is enclosed in curly brackets and prefixed with the keyword `atomic`, this indicates that the sequence is to be executed as one indivisible unit, non-interleaved with other processes. In the interleaving of process executions, no other process can execute statements from the moment that the first statement of an atomic sequence is executed until the last one is completed.

For instance, the following piece of code is used to swap the values of `a` and `b` by applying the `atomic` feature:

Listing 5.3.4 *Swapping two variables in Promela.*

```
atomic {
    tmp = b;
    b = a;
    a = tmp
}
```

Basic uses of Promela have been introduced in the above parts. The operational semantics of this modeling langauge may be found in [Hol03, GMP04, NH96b, Wei97b]. To understand more about Promela, please refer to [BA08, Hol03].

5.3.2.3 LTL Formulas

In SPIN, specifications for constraints are written in LTL, which allows qualitative describing and reasoning about changes of the truth values over time. LTL is based on the propositional calculus and consists of propositional variables, logical operators and temporal operators. In essence, a temporal logic allows for the specification of a relative order of events and is an appropriate choice for expressing behavior of the execution of reactive systems [BK08].

The logical operators are displayed in Table 5.1 and the temporal operators are shown in Table 5.2:

Operator	**Notation**
not	!
and	&&, $\wedge$
or	||, $\vee$
implies	$->$
equivalent	$<->$

Table 5.1: Logical operators used in LTL.

Operator	**Notation**	**Meaning**
always	[ ]	The formula [ ]ϕ holds, if ϕ holds for every state.
eventually	<>	The formula <>ϕ holds, if ϕ eventually occurs at some state.
until	U	The formula ϕ U ψ holds, if ϕ holds until ψ occurs.

Table 5.2: Temporal operators used in LTL.

Several examples of expressions/formulas in LTL:

- The formula $[\,](x >= 10)$ states that `x` should always be no less than 10.
- The formula $[\,](\neg(passport \vee ticket) -> \neg board\_flight)$ indicates the case that if the passenger does not hold either the passport or the ticket, he or she cannot take the flight.
- The formula $[\,](received -> <> processed)$ states that if the request is already received, finally it will be processed.
- The formula $[\,](\neg critical_1 \vee \neg critical_2)$ indicates the property for the mutual exclusion problem, i.e. at least one of the two processes is not in the critical section at one time.

5.3.2.4 Example

To illustrate the use of SPIN and LTL for verification and validation purpose, let us consider mutual exclusion, which refers to the problem of ensuring that no two processes can enter the critical section at the same time. Two programs trying to solve the mutual exclusion problem (taken from [Hol03]) are given below. The variable `cnt` is used to count the number of processes in the critical section. We can verify with the LTL formula $[](cnt <= 1)$, which means that no more than one process can enter the critical section at one time.

The program shown in Listing 5.3.5 gives a wrong solution. When verified with SPIN, the output shown below indicates the violation for a wanted assertion, which means that there may be two processes in the critical section at the same time (i.e. the variable `cnt` may be larger than 1):

```
pan:1: assertion violated !( !((cnt<=1))) (at depth 138)
pan: wrote wrongMutual.trail
```

Listing 5.3.5 *Faulty mutual exclusion algorithm.*

```
byte cnt;
byte x, y, z;
active [2] proctype user(){
    byte me = _pid + 1;    /* me is 1 or 2 */
    again:
    x = me;
    if
        :: (y == 0 || y == me) -> skip
        :: else -> goto again
    fi;

    z = me;
    if
        :: (x == me) -> skip
        :: else -> goto again
    fi;

    y = me;
    if
        :: (z == me) -> skip
        :: else -> goto again
    fi;

    /* enter critical section */
    cnt++;
    /* exit critical section */
    cnt--;
    goto again
}
```

The correct solution is shown below (see Listing 5.3.6). When verified with SPIN, no error occurs.

Listing 5.3.6 *Peterson's mutual exclusion algorithm.*

```
bool turn, flag[2];
byte cnt;
active [2] proctype P1() {
    pid i, j;
    i = _pid;
    j = 1 - _pid;

    again:
    flag[i] = true;
    turn = i;
    (flag[j] == false || turn != i) -> /* wait until true */

    /* enter critical section */
    cnt++;
    /* exit critical section */
```

```
16      cnt--;
17
18      flag[i] = false;
19      goto again
20  }
```

To understand more about LTL, interested readers may refer to [KM08, Roz11].

5.3.3 INICheck

To support programmers to detect possible errors when writing INI programs, we develop a tool called INICheck (available to download at [LP12]) to automatically convert a significant subset of INI to Promela in order to model-check it.

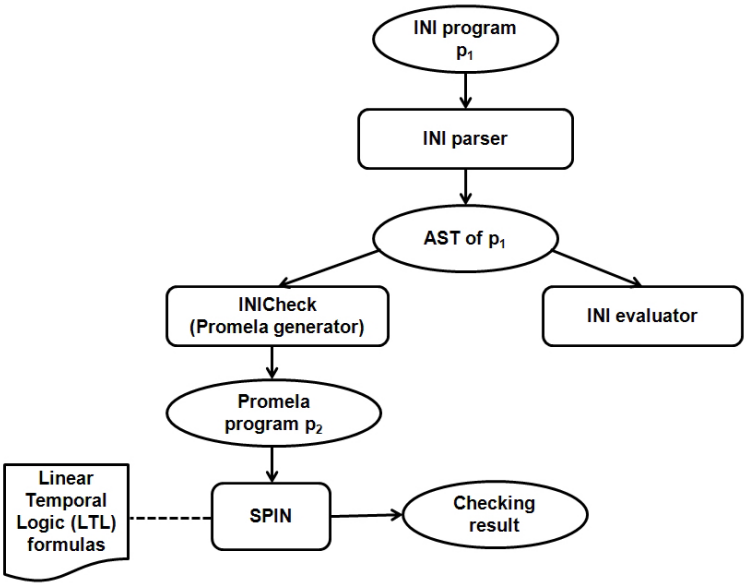


Figure 5.3: Overall approach for model checking INI programs.

There are several key reasons for selecting SPIN in our approach:

- INI and Promela have similarities. In both languages, programmers may express rules with guards (i.e. conditional expressions) that need to be fulfilled before the upcoming action can be executed.
- Both INI and Promela support concurrency. In INI (Promela), events (processes) can run in parallel.

- SPIN is a good tool to verify concurrent systems [BA08].

- SPIN has been applied widely in both academy and industry [GHJX08, HB07, KCKK08]. In estimation, between 5,000 and 10,000 people are routinely using SPIN [RH04].

Since Promela is a modeling language, clearly not all features of INI can be converted. Moreover, as not all INI elements are meaningful for modeling purpose, we focus only on essential ones. As shown in Figure 5.3, INICheck generates a Promela program from the AST of an INI program. Specifically, an AST walker checks each AST node, maps and translates it to the corresponding element structure in Promela. Our tool can fully handles the following INI elements: function declarations, function invocations, rules, built-in events, event synchronization mechanism, the `case` statement, arithmetic expressions, logical expressions and variable assignments. We sum up the commonalities between INI and Promela in Table 5.3, and then the conversion rules are discussed in the next parts.

	INI	**Promela**
Data types	Support basic data types and user-defined ones.	Support numeric data types and user-defined ones.
Statements	Has conditional statement and repetitive structures (with rules or events).	Has conditional and repetitive statements.
Variables	All variables are global inside the function.	Has global and local variables.
Concurrency	Events may run in parallel.	Processes may run in parallel.

Table 5.3: Comparison between INI and Promela.

Converting Functions Each function is converted to a proctype having the same name. The proctype named `main`, which corresponds to the function `main` in INI programs, is invoked inside the `init` proctype of Promela. Listing 5.3.7 shows the conversion on a sample INI program (on the left) into a Promela program (on the right).

Listing 5.3.7 *Converting functions into proctypes.*

```
function main() {
    ...
    foo(5);
    ...
}
function foo(n) {
    ...
}
```

```
...
proctype main() {
    ...
    run foo(5);
    ...
}
proctype foo(int n) {
    ...
}
init {
    run main();
}
```

Converting Variables We can convert integer and boolean data types from INI to Promela. Initialized values for variables also are kept. Listing 5.3.8 displays the conversion on a sample program.

Listing 5.3.8 *Converting variables.*

```
function main() {
    @init() {
        i = 1
        j = true
    }
    ...
}
```

```
int i = 1
bool j = true
proctype main() {
    ...
}
init {
    run main();
}
```

Converting Statements Promela only supports a few kinds of statements: assignment statement, `do`-statement, `if`-statement, `printf`-statement, `goto`-statement. Therefore, not all statements in INI have a counterpart in Promela like the case of the `return` statement. In Table 5.4, we show a map among statements in the two languages INI and Promela for conversion purpose.

Statement in INI	**Statement in Promela**
Assignment statement	Assignment statement
`Case` statement	`If` statement
`Print` or `Println` statement	`Printf` statement

Table 5.4: Mapping statements between INI and Promela.

Converting Rules Each rule is converted to a branch of a `do`-statement in Promela since both of them can be used to express a repeated action. The logical expression part of an INI rule becomes a guard statement in Promela. As stated earlier, the INI evaluator checks the rules sequentially. However, in the `do`-statement in Promela, if more than one branch is executable, one of them will be selected non-deterministically. We can recognize that SPIN handles not only execution paths for rules in INI but also other arbitrary ones. Therefore, in case of using rules in INI and a counterexample is found with SPIN, we may need to check whether the execution path leading to the error follows INI specification or not (double-check process). In other words, a counterexample found with SPIN may not be reachable in INI, but the converse holds: every execution path of INI is covered in the translated Promela code.

Listing 5.3.9 shows the conversion on a sample INI program, in which we have two rules (lines 5-7 and lines 8-10). These rules are converted to two branches of a `do`-statement in a Promela program (lines 4-5).

Listing 5.3.9 *Converting rules.*

```
1 function main() {
2     @init() {
3         i = 1
4     }
5     i == 1 {
6         ...
7     }
8     i == 2 {
9         ...
10    }
11 }
```

```
1 int i = 1
2 proctype main() {
3     do
4         :: i == 1 -> ...
5         :: i == 2 -> ...
6     od;
7 }
8 init {
9     run main();
10 }
```

Converting Events Currently, all built-in events can be fully automatically converted. The `@init` event in INI is mostly used to initialize necessary variables, and therefore it can be embedded into the initialization part of a Promela program. Other events are converted to corresponding proctypes in Promela. For example, each timing event like `@every` or `@cron` is converted to one proctype with a `do`-statement inside (to express the repetition of an action). The `@update` event is converted to one proctype along with a channel for the notification of changes on the observed variable. If there is an `@end` event in our INI

program, each other event when converted will have one associated variable, which is set to `true` after the event action terminates. When all these variables become `true`, including one for the proctype `main` (i.e. all the corresponding events and rules are terminated), the proctype corresponding to the event `@end` can run.

Regarding user-defined events, INICheck only preserves the structure of the action coming with each of them. In some cases, this is enough for modeling and verification purpose.

Listing 5.3.10 illustrates the conversion on a sample INI program with the use of an `@every` event. The `@every` event at lines 3-5 is converted to the proctype named `every` at lines 4-8.

In another example, the `@update` event in Listing 5.3.11 is converted into the corresponding proctype and channel in Listing 5.3.12. Note that all temporary variables are automatically created.

Listing 5.3.10 *Converting the `@every` event.*

```
1 function main() {
2     ...
3     @every[time=1]() {
4         ...
5     }
6     ...
7 }
```

```
1  proctype main() {
2      ...
3  }
4  proctype every() {
5      do
6          :: true -> ...
7      od;
8  }
9  init {
10     atomic {
11         run main();
12         run every();
13         ...
14     }
15 }
```

Listing 5.3.11 *Using `@update` event in an INI program.*

```
1 function main() {
2     @init() {
3         v = 0
4     }
5     @update[variable=v](oldv,newv) {
6         ...
7     }
```

```
8       @every[time = 1000]() {
9           v++
10      }
11  }
```

Listing 5.3.12 *Using channel to notify when a variable is updated in Promela.*

```
1  int v = 0
2  int temp1_v = 0
3  int temp2_v = 0
4  chan chan_v= [0] of {int}
5  proctype update_v() {
6      do
7          :: chan_v ? temp2_v->
8              if
9                  ::(temp2_v != temp1_v)->
10                     temp1_v = v;
11                     ...
12                 :: else -> skip
13             fi;
14
15     od;
16 }
17 proctype every(){
18     do
19         :: true ->
20             v++;
21             chan_v ! v;
22     od;
23 }
24 init {
25     atomic {
26         run every();
27         run update_v();
28     }
29 }
```

If there is an `@end` event in our INI program, for instance, if an INI program has two events `e1` and `e2`, along with the `@end` event, the corresponding Promela program is shown in Listing 5.3.13:

Listing 5.3.13 *Handling the `@end` event when converting to Promela.*

```
1  bool endE1 = false
2  bool endE2 = false
3  proctype e1() {
4      do
```

```
        ...
    od;
    endE1 = true;
}
proctype e2(){
    do
        ...
    od;
    endE2 = true;
}
proctype end() {
    do
        :: (endE1 && endE2) -> ...
        :: else -> skip;
    od;
}
init {
    atomic {
        run e1();
        run e2();
    }
    run end();
}
```

Converting the Synchronization Mechanism Assume that a given event `e0` synchronizes on other events `e1, ..., eN`. When we convert this relationship to Promela, to deal with mutual exclusion problem among possible multiple event threads, we create a temporary boolean variable for each event, that is set to `true` as soon as the event is running. We use them to handle synchronization through an `atomic` statement waiting for all the variables to be `false` at the same time, as shown in Listing 5.3.14. This blocking approach is also described in [BA08].

Listing 5.3.14 *Converting the synchronization mechanism in INI to Promela.*

```
...
bool e0Sync = false
bool e1Sync = false
...
bool eNSync = false
proctype e0() {
    do
        ::atomic { e1Sync == false && e2Sync == false ...
         && eNSync == false
            -> e0Sync = true;
        }
```

```
12          ::e0Sync == true
13              -> ... e0Sync = false;
14      od;
15  }
16  proctype e1() { ... }
17  ...
18  proctype eN() { ... }
19  init {
20      atomic {
21          run e0(); run e1(); ... run eN();
22      }
23  }
```

5.3.4 Examples

In this section, we show some examples of using INICheck to analyze and verify properties and constraints.

5.3.4.1 Detecting Infinite Loops

Detecting infinite loops in software programs has been a hot topic in research on program comprehension and analysis [BJSS09, CMKR11, LS09]. With the tool INICheck and using `acceptance` labels to find acceptance cycles in SPIN, we can detect infinite loops in INI programs. For example, let us consider an INI program on the left of Listing 5.3.15. Clearly, the two rules of our program will run infinitely since after `v` is set to `1`, it will be set to `2` and vice versa. The conversion of this program in Promela is shown on the right of Listing 5.3.15.

Listing 5.3.15 *An INI program containing infinite loops and the corresponding Promela program.*

```
function main() {
    @init() {
        v = 1
    }
    v == 1 {
        v = 2
    }
    v == 2 {
        v = 1
    }
}
```

```
1  int v = 1
2  proctype main(){
3      do
4          :: (v == 1) ->
5              accept_1:
6              v = 2;
7          :: (v == 2) ->
8              accept_2:
9              v = 1;
10         :: else -> break;
11     od;
12 }
13 init {
14     run main();
15 }
```

As described in Section 5.3.2, specifications in SPIN can be expressed as assertions, labels, never claim, or LTL formulas. In this case, we use labels with the prefix `accept`. Each branch of the `do`-statement is therefore labeled with a corresponding label `accept_n` (line 5 and line 8). In SPIN, an acceptance-state is a state in which some process instances are at a statement labeled with the prefix `accept`. When checking for acceptance cycles, the verifier will complain if there is an execution visiting infinitely often an acceptance-state [Ger97]. As a result, checking acceptance cycles allows us to detect infinite loops. Running with SPIN (version 6.1.0) finds an infinite loop as shown below, and this confirms our judgement.

```
pan:1: acceptance cycle (at depth 1) ...
State-vector 28 byte, depth reached 12, errors: 1 ...
```

5.3.4.2 Detecting Unreachable Code

There may exist unreachable code (dead code). Detecting it is essential to help us analyze and simplify programs [BSS12, CCK11]. In an INI program on the left of Listing 5.3.16, the action defined in the rule `v==6` (lines 8-10) is never executed since `v` is never set to `6` (the maximum and also final value of `v` is 5, see the rule at lines 5-7). As a result, line 9 is unreachable.

Listing 5.3.16 *An INI program containing unreachable code and the corresponding Promela program.*

```
function main() {
  @init() {
    v = 1
  }
  v < 5 {
    v++
  }
  v == 6 {
    v = v+2
  }
}
```

```
int v = 1
proctype main() {
  do
    ::v < 5 -> v++;
    ::v == 6-> v = v+2;
    ::else -> break;
  od;
}
init {
  run main();
}
```

After converting our program to Promela (on the right of Listing 5.3.16), SPIN detects unreachable code as shown below.

```
...
unreached in proctype main
    UnreachableCode.pml:5, state 4, "v = (v+2)"
    (1 of 10 states)
...
```

Another example of using INICheck for verifying desired constraints and properties for events (expressed in LTL) will be introduced in Section 6.1.3, page 157.

5.4 Conclusion

In this chapter, we introduced INI's type system, with main focuses on how programmers may define and use both built-in and algebraic data types, and how the type checking engine in INI works. INI has a strong type system, which is appropriate for building context-aware reactive and self-adaptive software. Furthermore, based on the commonalities between INI and Promela, a tool called INICheck was developed for conversion purpose. Some examples also were shown to illustrate the ideas of model checking INI programs by combining INICheck and SPIN.

Using INICheck may help programmers to statically analyze properties of their programs and verify wanted behavior. Since some program structures in Promela are nondeterministic, the converted Promela program will cover not only execution paths in INI code

but also other arbitrary ones. Therefore, all properties and errors in an INI program will be reflected in the corresponding Promela program, and our approach is complete. However, a problem found with Promela may not be reproducible with INI. In other words, when a counterexample is found, programmers need to check all steps leading to the violation through the simulation mode of SPIN, and then examine whether this behavior can be obtained in INI or not. In case of converting rules, we could refine our approach by introducing variables to simulate a rule counter, similarly to the operational semantics `[R,i,j]` (see Section 4.2.2.2 of Chapter 4, page 100 for more details). For instance, each rule R_i can be translated to two branches of a `do`-statement in Promela:

```
::r == i && cond -> action; r++
::r == i && !cond -> r++
```

in which `r` is the an extra index used to indicate which rule is currently evaluated, `i` is the index of a rule, and `cond` and `action` are its corresponding condition and action. However, by leaving room for nondeterministic choice, we are ensured to be consistent with other possible INI implementations, where the rule evaluation order can be different.

Our implementation for converting INI programs to corresponding Promela programs still has several limitations as listed below:

- At present, parameters of events (except for the event `@update`) are ignored during conversion. This conversion will be needed when the parameters are meaningful for modeling an INI program.

- The translation of user-defined events still needs human intervention or proofreading to ensure the correctness, since each one has its own semantics specified by programmers. A full automation, for this very reason, is out of reach.

- Our tool still does not enable multiple occurrences of the same event. This can be done by assigning an integer counter for each converted proctype and then applying a similar execution strategy as with rules. Additionally, in this case, the conversion for the synchronization mechanism also needs to be updated, in which we may reuse the integer counters for expressing prerequisites instead of using boolean variables as before (see page 147).

- Currently, timing is ignored when converting from INI to Promela. Nevertheless, in case that timing is essential for modeling an INI program, we need to support the conversion of it, although SPIN does not easily support the modeling of such a notion.

Clearly, as in other related work (converting Java, C, or Ada to Promela [Hav99, Ke09, FMSP11]), a comprehensive automatic conversion still requires more work to obtain. The main reason is that Promela is a modeling language and it has fewer features than a regular language. For example, it does not support I/O operations, network programming, data types for real numbers, etc. Moreover, many INI functions, especially complex ones, do not have counterparts in Promela.

Chapter 6

Case Studies

In this chapter, we present two case studies of using INI for programming context-aware and event-based embedded systems. The first one is a M2M gateway program that is developed and tested on a real industrial device inside the scope of the MCUBE project. The other one is an object tracking program running on the humanoid robot Nao.

6.1 A Multimedia M2M Gateway

6.1.1 Developing a Multimedia M2M Gateway Program with INI

Many M2M applications are required to send data to a M2M server through a network and a M2M gateway is a typical example of this architecture [SH11]. Normally, a M2M gateway allows different types of networks to communicate with each other in order to provide data [Sos09]. Users may utilize this device for data collection or for surveillance purpose [Den12]. For instance, in photovoltaic energy, M2M applications are used to capture images to track the cleanness of solar panels so that leaves or other garbage can be detected. Another example is on the streets, where vehicular and pedestrian traffic can be watched so that drivers can avoid congested roads and the local government can find the way to ameliorate the transport system. In a factory, M2M sensors are used to track and monitor assets, equipment, materials, cargo and supplies. Since this kind of tasks does not create much added-value and may be in dangerous or remote environments, taking advantages of M2M technologies is a good solution.

We develop a multimedia M2M gateway program implemented inside the scope of the

MCUBE project [FED12]. Our INI program (adapted from [LHM+13]) contains two basic steps as shown in Figure 6.1:

- Collecting regularly multimedia data (e.g. images, sound, etc.), which are captured by sensors or peripherals.
- Transmitting data by schedule to the server through the network for repository purpose and further energy-consuming treatments.

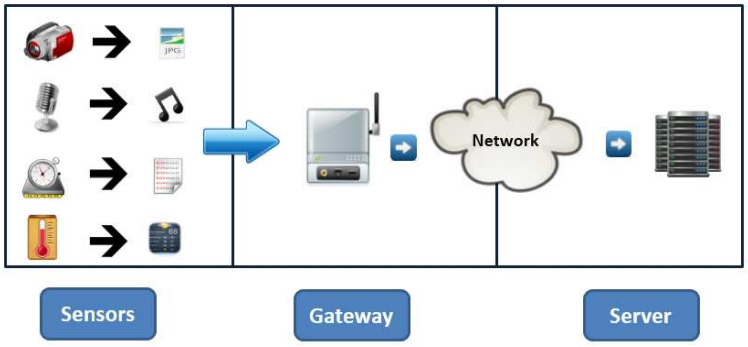


Figure 6.1: The role of a gateway.

All these operations above can be scheduled easily with the help of the two built-in events `@every` and `@cron` (see Table 3.1, page 68). The event `@every[time:Integer]()` does an action periodically. The event `@cron[pattern:String]()` occurs on times indicated by the UNIX CRON pattern expression (see Section 3.2.3.1, page 68).

Listing 6.1.1 *A M2M gateway program written in INI.*

```
function main() {
    //Initialization
    @init() {
        capturedDataFolder = file("data")
        //Create a new data folder if it does not exist
        case {
            !file_exists(capturedDataFolder) {
                mkdirs(capturedDataFolder)
            }
        }
        zipFile = file("data.zip")
        keepParentFolder = true
    }

```

```
15      //Capture image from the camera using the library gphoto2
16      e1:@every[time = 60000]() {
17          exec("gphoto2␣--capture-image-and-download␣--filename␣"
18           + "data/img" + time() + ".jpg")
19      }
20
21      //Capture sound from the microphone using the library alsa
22      e2:@every[time = 30000]() {
23          exec("arecord␣-d␣30␣-f␣cd␣" + "data/sound" + time()
24           + ".wav")
25      }
26
27      //Schedule an upload of data to the FTP server
28      @cron[pattern = "0␣09-18␣*␣*␣1-5"]() {
29          stop_event([e1,e2])
30          zip(capturedDataFolder,zipFile)
31          upload_ftp("server_address", "user_name",
32           "password", zipFile, to_string(time()) + "data.zip")
33          delete_file(zipFile)
34          delete_folder(capturedDataFolder, keepParentFolder)
35          restart_event([e1,e2])
36      }
37  }
```

Our complete program is shown in Listing 6.1.1. The `main` function is composed of four events. The event `@init` (lines 3-13) is invoked to initialize necessary variables that will be used later. The variable `capturedDataFolder` indicates the folder where we put the collected data. If this folder does not exist, we create it (lines 7-9). The variable `zipFile` denotes a zipped data file. We compress the data before uploading to save network bandwidth, which is a major cost, particularly when using 3G/4G connection, compared to compression cost.

The next two events are `@every` event kind. They are used to collect image and sound data (e.g. in a field or in a factory). The event `e1` (lines 16-19) is invoked every minute to get a picture captured by a camera (by using the library `gphoto2` [gPh13]). The event `e2` (lines 22-25) is invoked every 30 seconds to record sound through a microphone (by using the library `alsa` [Als12]). All files (i.e. pictures and sounds) are saved into the data folder. These collecting processes (i.e. two events) run in parallel to take advantage of multithreading for better performance.

The last event is a `@cron` event (lines 28-36), which is employed to upload the data to a File Transfer Protocol (FTP) server every hour during working hours (i.e. 9:00 AM -

6:00 PM) on every weekday (i.e. from Monday to Friday). Inside this event, first, we stop the two events `e1` and `e2` (line 29). Next, we compress the data folder before uploading (line 30). Then we upload the compressed data to a FTP server (lines 31-32). Since the gateway is only a data-exchanging device with a limited storage capacity, after uploading, we delete the data to save storage (lines 33-34). Finally, we restart the two events `e1` and `e2` so that we can collect data again (line 35).

6.1.2 Testing Results and Evaluation

We tested our program on a real industrial gateway. The device is provided by Webdyn [Web13], a company dedicated to the design, development, and marketing of this kind of product and also one of our partners in the MCUBE project. We used Oracle Java SE Embedded for establishing the runtime environment of INI on the gateway. This platform is optimized for mid-range to high-end embedded systems and offers a high-performance virtual machine, full high-performance graphics support, deployment infrastructure, and a rich set of features and libraries [HTW12, Ora13b].

During all the experiments, our program worked well. All the data were captured and transmitted properly. One remarkable thing is that it is easy to extend our program to work with multiple sensors concurrently. For example, it may work with several cameras oriented in different directions.

Now let us compare our INI program with a Java program that would perform the same tasks. In order to make all operations running in parallel in Java, we need to create some explicit threads for different tasks: capturing pictures, recording sounds, uploading data and time scheduling for all operations. Scheduling is a nontrivial task to implement in Java. In addition, we also need some thread pools in order to manage and synchronize those threads when needed. Although Java has a powerful support for concurrency, writing correct concurrent applications in this language is still challenging and error-prone, especially for novice programmers [GP06]. To decrease the difficulty and hide the unnecessary complexity, INI separates the thread issue (a part of the interpreter), and the event-handling issue (implemented in INI). This separation of concerns helps in making the INI approach clearer and less fallible than a pure-Java program. Furthermore, if

the code were written in C/C++, this still requires more time and effort. For instance, Webdyn has built five APIs in C/C++ for their own system just for scheduling purpose: `pf_add_daily_schedule`, `pf_add_weekly_schedule`, `pf_add_monthly_schedule`, `pf_add_yearly_schedule`, and `pf_add_oneshot_schedule`. In this case, in order to switch among different schedules, programmers have to recompile the whole application, while a program in INI needs only to call the `reconfigure_event` primitive. Therefore, using INI brings us many benefits for developing M2M systems when compared with other classical programming languages.

6.1.3 Model Checking a Prototype of the M2M Gateway

In this part, we show how to apply our tool INICheck (see more in Section 5.3.3, page 141) and the model checker SPIN to verify some desired properties on a prototype (a simplified version) of the M2M gateway. This prototype is developed before implementing our complete program shown in Listing 6.1.1 (page 154) so that we can test and check initially some properties.

Listing 6.1.2 *A prototype of the M2M gateway written in INI, which detects moving balls, collects, compresses and then uploads data.*

```
function main() {
    @init() {
        dataFile = file("ballData.csv")
        zipFile = file("ballData.zip")
        isZipped = false
        isUploading = false
    }
    $(e) b:@ballDetection[period=1000](r,x,y) {
        isZipped = false
        case {
            !file_exists(dataFile) {
                create_file(dataFile)
            }
        }
        fwriteln(dataFile,to_string(time()) + ",␣" + r + ","
         + x + ",␣" + y)
        zip(dataFile, zipFile)
        isZipped = true
    }
    $(b) e:@every[time = 5000]() {
        case {
            isZipped {
```

```
23                  isUploading = true
24                  upload_ftp("host", "username", "password",
25                   zipFile, to_string(time()) + "bdUpload.zip")
26                  delete_file(dataFile)
27                  delete_file(zipFile)
28                  isUploading = false
29              }
30          }
31      }
32  }
```

Our INI program (Listing 6.1.2) uses a video camera to detect the movement of a ball, get its up-to-date position in space periodically, saves it into a CSV file, and finally upload the compressed data file to a FTP server. In this program, we define three events. The event `@init` initializes four variables. `dataFile` and `zipFile` specify the data file and zipped file. `isZipped` is used to indicate whether the data file is already compressed or not and `isUploading` is used to indicate whether data is being uploaded or not (those two variable are merely used to express the property we want to check as shown later). We compress the data file in order to save network bandwidth. The user-defined event `@ballDetection` (identified by `b`) is used to detect the ball and send its position to the program when detected. This event has one input parameter called `period`, which is applied to set how long the event should sleep between two image detections (time unit is in milliseconds). Besides, we have three output parameters `(r,x,y)`, which are the radius and coordinates of the detected ball in the captured image (see more about this user-defined event in Section 3.2.3.2). The last event is `@every` (identified by `e`), which is applied to upload the compressed data file periodically. It is essential that when we collect data (`@ballDetection`), the uploading process (`@every`) does not run and vice versa. In other words, these two events must be mutually exclusive. As a result, `b` and `e` need to be mutually synchronized, as it is done at lines 8 and 20.

Listing 6.1.3 *The Promela model for a prototype of the M2M gateway shown in Listing 6.1.2.*

```
1 bool isZipped = false
2 bool isUploading = false
3 bool bSync = false
4 bool eSync = false
5 proctype ballDetection() {
```

```
6       do
7           ::atomic { eSync==false -> bSync = true; }
8           ::bSync == true ->
9               isZipped = false;
10              isZipped = true;
11              bSync = false;
12      od;
13  }
14  proctype every() {
15      do
16          ::atomic { bSync==false -> eSync = true; }
17          ::eSync == true ->
18              if
19                  ::isZipped ->
20                      isUploading = true;
21                      isUploading = false;
22              fi;
23              eSync = false;
24      od;
25  }
26  init {
27      atomic { run ballDetection(); run every(); }
28  }
```

We want to ensure that before uploading, if there is new data, the data file must have been previously zipped. Listing 6.1.3 shows the automatic translation of this gateway program in Promela. During translation, two variables `bSync` and `eSync` are automatically created to synchronize the two proctypes `ballDetection` and `every` as explained earlier in page 147.

After conversion, we check the LTL formula `[](!isUploading || isZipped)`. Its meaning is that at any time, if the variable `isUploading` is `true`, the variable `isZipped` also must be `true`. When verifying the Promela code with this formula in SPIN, we find no error as shown below:

```
...
State-vector 24 byte, depth reached 24, errors: 0
...
```

Now if we remove synchronization for the two events `@ballDetection` and `@every` in our INI program, we get the Promela program as shown in Listing 6.1.4. Checking with SPIN now raises an error as below, which means that keeping synchronization to avoid

unwanted arbitrary interleaving between events is essential to ensure the correctness of our INI program.

```
...
pan:1: end state in claim reached (at depth 22)
...
State-vector 24 byte, depth reached 24, errors: 1
...
```

Listing 6.1.4 *The Promela program modeling a prototype of the M2M gateway shown in Listing 6.1.2, in case not using the synchronization mechanism.*

```
bool isZipped = false
bool isUploading = false
proctype ballDetection() {
    do
      ::true ->
        isZipped = false;
        isZipped = true;
    od;
}
proctype every() {
    do
      ::true ->
        if
          ::isZipped ->
            isUploading = true;
            isUploading = false;
        fi;
    od;
}
init {
    atomic { run ballDetection(); run every(); }
}
```

6.2 Tracking an Object with INI and Nao

6.2.1 Overview of Robot Programming

The word "robot" was coined by the Czech novelist Karel Capek in a 1920 play titled Rassum's Universal Robots. In Czech, "robot" means worker or servant. According to the definition of the Robot Institute of America dating back to 1979, a robot is:

> *"A reprogrammable, multifunctional manipulator designed to move material, parts, tools or specialized devices through variable programmed motions for the performance of a variety of tasks".*

At present, people require more from robots, since they are considered as a subset of "smart structures" – engineered constructs equipped with sensors to "think" and to adapt to the environment [Wad07]. Generally, robots can be put into three main categories: manipulators, mobile robots and humanoid robots [RN09], in which the number of lifelike humanoid robot has grown significantly in recent years [BCHM09]. In our case study, we also work with a humanoid robot.

Robots now play an important role in many domains. In manufacturing, robots are used to replace humans in remote, hard, unhealthy or dangerous work. They will change the industry by replacing the Computer(ized) Numerical(ly) Control(led) (CNC) machines. In hospitals, they are employed to take care of the patients, and even do complex work like performing surgery. In education, robots may be good assistants for the children. They may also be good friends for old people at home for talking and sharing housework. The global service robotics market in 2011 was worth $18.39 billion. According to one recent report, this market is valued at $20.73 billion in 2012 and expected to reach $46.18 billion by 2017 at an estimated Compound Annual Growth Rate (CAGR) of 17.4% from 2012 to 2017 [MM12]. As a result, research on robots gets increasing interest from governments, companies, and researchers worldwide [BCHM09, Bek08].

Building robot programs is a complex task since a robot needs to quickly react to variabilities in the execution environment. In other words, a robot should be indeed autonomous. Besides, it should be able to do several things at one time. Consequently, using a programming language or development framework dedicated to robots is essential. The ultimate goal is to help programmers develop robot programs more efficiently and straightforwardly. A general robot programming paradigm is shown in Figure 6.2. Typically, a program sends commands to a robot to observe things in the work space, and captures data through sensors. Based on the values received from the sensors, a program may control the robot to react to changes.

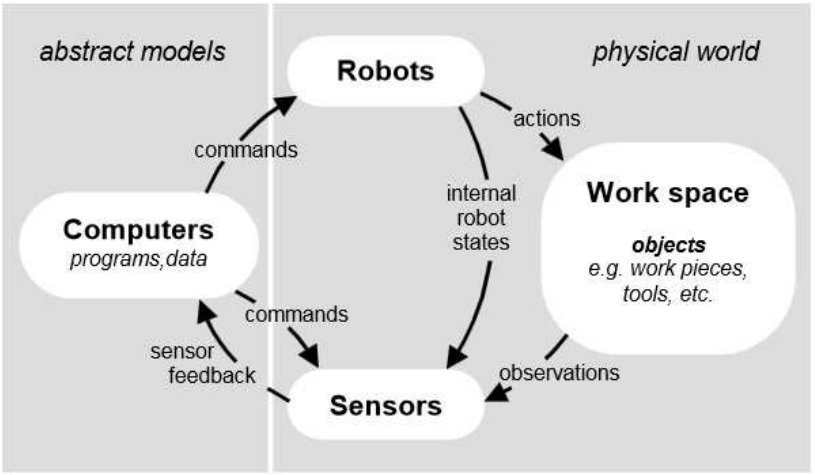


Figure 6.2: General robot programming paradigm [WT10].

Classical languages like Java, C/C++, .Net are usually used for programming robots [Auy06, KGCC11, Pre05]. However, developing robotic applications using these languages requires more time and effort since they are not fully dedicated to this purpose. For example, to support interaction between robots and an environment when something happens, programmers have to construct a mechanism for event detection and handling. In order to do this, programmers need to write a lot of code or use some extensions like [JE09, Rob13], although they are not easy to adapt.

Additionally, several robotic development platforms and DSLs have been designed to assist programmers. Urbiscript is a scripting language primarily designed for robotics that is already introduced in Section 2.2. The KUKA robot programming language is developed by KUKA, one of the world's leading manufacturers of industrial robots [Cor13]. KUKA is simple, Pascal-like and lacks many features. Since using KUKA has some limitations, a Matlab abstraction layer has been introduced to extend its capabilities [CSMP11]. RoboLogix [Inc13] is a scripting language that utilizes common commands, or instruction sets among major robot manufacturers. RoboLogix programs consist of data objects and a program flow. The data objects reside in registers and the program flow represents the list of instructions, or the instruction set, which is used to program the robot. However, RoboLogix still does not supply a well-defined mechanism for the robots to interact and react to the environment.

6.2.2 Introduction to Nao

Nao is a humanoid robot that is built by the French company Aldebaran-Robotics [GHB+08, Ald13b]. In general, humanoids are "robots that are clearly seen as machines but have human characteristics such as a head (with no facial features), a torso, arms, and possibly legs" [BCHM09]. Nao is equipped with many sensor devices to obtain close environment information (see Figure 6.3). It has for instance become a standard platform for RoboCup, an international initiative that fosters research in robotics and artificial intelligence [Fed13].

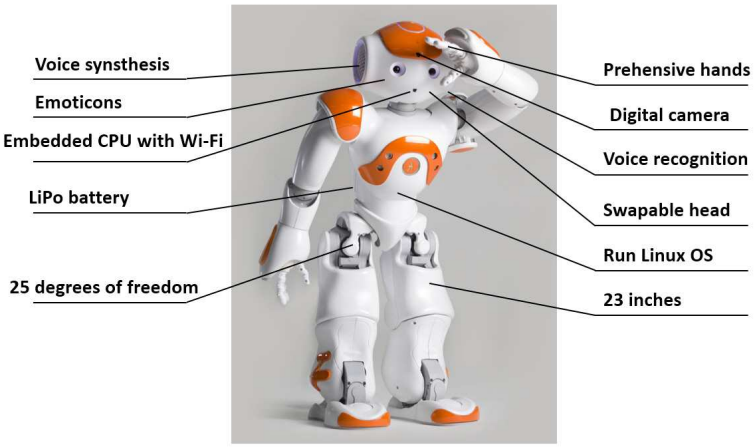


Figure 6.3: Nao's features [Ald13b].

Nao comes with two software as shown in Figure 6.4.

- **Embedded software**: running on the motherboard located in the head of the robot, allowing an autonomous behavior. The operating system of the robot is OpenNao. NAOqi is the middleware running on Nao that helps prepare modules to be run either on Nao or on a remote PC. Code can be developed on different operating systems like Windows, Mac or Linux, and be written in many languages including C++, Java, Python and .Net. The company Aldebaran Robotics developed many modules built on top of this framework that offer rich APIs for interacting with Nao, including functionalities related to audio, vision, motion, communication or even several low-level accesses. They also provide a well-organized documentation, particularly on how to control the robot effectively [Ald13a].

- **Desktop software** (named Choreographe): running on normal computers, allowing the creation of a new behavior, and the remote control of the robot.

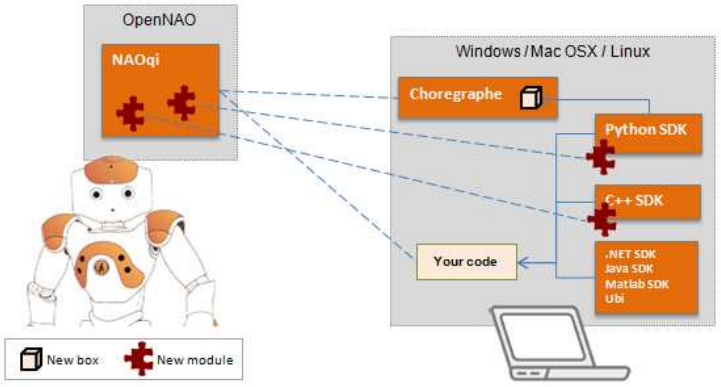


Figure 6.4: Controlling Nao [Ald13b].

6.2.3 Implementing an Object Tracking Program

In this part, we show how INI can be applied for programming robot through the example of building a ball tracking program (adapted from [LFH+13]). Cognition and navigation are important must-have features for mobile robots in general and humanoid robots in particular [Pat07].

6.2.3.1 Overall Ideas

In our program, we want to control the robot to:

- Detect a ball in the space.
- Walk to reach the ball (if detected).

Figure 6.5 displays the possible relative positions between the robot and the ball. There are three distinguished zones that are specified based on the calculated distance between them. And then according to which zone the ball belongs to, we can control the robot and adopt the desired behavior:

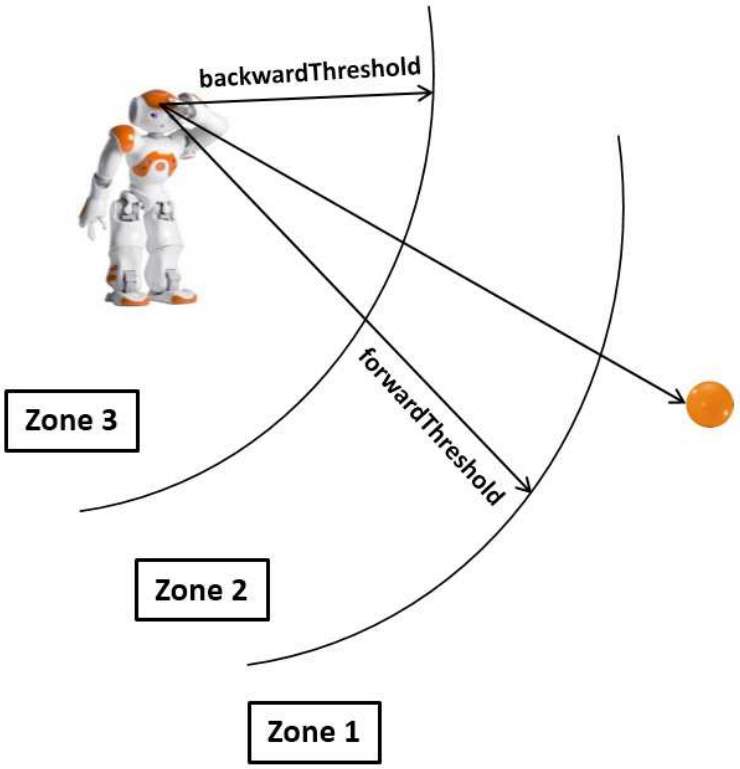


Figure 6.5: Possible relative positions among the robot and the ball.

- Zone 1: When the distance from the robot to the detected ball is larger than the `forwardThreshold` (unit is in meters and its range is 0.0 - 1.0 meters), the ball is considered as far from the robot and it needs to move in order to reach the ball.

- Zone 2: When the distance from the robot to the detected ball is between the two thresholds, `backwardThreshold` (unit and range are the same as `forwardThreshold`) and `forwardThreshold`, the robot does not move since its current place can be considered as a good position to observe the ball. However, the robot's head still can turn to continue to follow the ball.

- Zone 3: When the distance from the robot to the detected ball is shorter than `backwardThreshold`, the ball is considered as too close and as moving towards the robot. As a result, the robot will go backward in order to avoid the collision and keep its eyes on the ball.

The activity diagram of the strategy is shown in Figure 6.6.

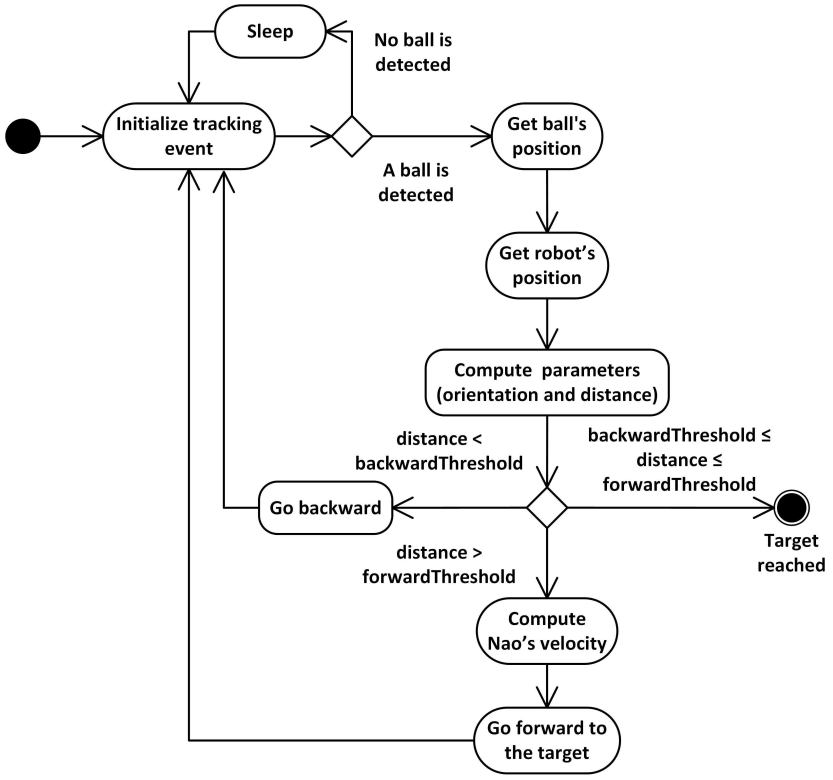


Figure 6.6: The activity diagram for an object-tracking program.

6.2.3.2 Implementation

Our program is shown in Listing 6.2.1. In our program, we employ three events. The event `@init` (lines 3-17) is applied to initialize the variables used later in our program. The purpose of the two variables `forwardThreshold` and `backwardThreshold` has been explained above. The variable `interval` (unit is in milliseconds) sets the delay after which, if no ball is detected, the robot temporarily stops tracking. The variable `stepFrequency` (normalized between 0.0 and 1.0, see more in [Ald13a]) is applied to set how often the robot will move and the variable `defaultStepFrequency` is applied to set the default value for step frequency. The two variables `ip` and `port` are used to indicate the parameters for Nao's network address. The boolean variable `useSensors` is used to indicate whether the program uses the direct returned values from sensors or the values after adjustments

by the robot itself (please refer to Nao's documentation [Ald13a] to understand more). The variable `targetTheta` is the robot orientation relative to the ball's orientation. The variable `robotPosition` points out the robot's position when it detects the ball so that then we can calculate appropriate needed direction and speed for its movement. The robot's position is specified by three parameters: `x`, `y` and θ. `x` is the distance along the `X` axis in meters (forwards and backwards). `y` is the distance along the `Y` axis in meters (lateral motion). θ is the robot orientation relative to the current orientation (i.e. the rotation around the `Z` axis) in radians [-3.1415 to 3.1415]. `stepX` is the fraction (between 0.0 and 1.0) of `MaxStepX` (the maximum translation along the `X` axis for one step, see [Ald13a]). The sign of `stepX` also indicates the moving direction (forward or backward) of Nao. The boolean variable `needAdjustDirection` is used to indicate whether we need to adjust the direction when the robot moves towards the ball. The intention of using the temporary variable `i` will be explained later.

Listing 6.2.1 *An object tracking program written in INI.*

```
function main() {
    //Initialization
    @init() {
        forwardThreshold = 0.5
        backwardThreshold = 0.3
        interval = 1000
        stepFrequency = 0.0
        defaultStepFrequency = 1.0
        ip = "nao.local"
        port = 9559
        useSensors = false
        targetTheta = 0.0
        robotPosition = [0.0,0.0,0.0]
        stepX = 0.0
        needAdjustDirection = false
        i = 0
    }

    //Detect a ball in space
    $(e) b:ballDetection[robotIP = ip, robotPort = port,
     checkingTime = interval](ballPosition){
        //Compute necessary parameters, and return in an array
        parameters = process_position(ip, port, ballPosition,
         forwardThreshold, backwardThreshold, useSensors)
        targetTheta = parameters[0]
        robotPosition = parameters[1]
```

```
27          stepX = parameters[2]
28          i = 0
29          needAdjustDirection = true
30          stepFrequency = defaultStepFrequency
31      }
32
33      //Control the robot periodically
34      $(b,e) e:@every[time = 200]() {
35          //Control the robot to go one step if the ball
36          //is detected
37          needAdjustDirection = reach_to_target(ip, port,
38           stepFrequency, robotPosition, stepX, targetTheta,
39          needAdjustDirection, useSensors)
40          i++
41          case {
42              //Reset parameters after three consecutive
43              //walking steps
44              i>3 {
45                  stepX = 0.0
46                  targetTheta = 0.0
47                  stepFrequency = 0.0
48              }
49          }
50      }
51  }
```

The event `@ballDetection` (lines 20-31) is a user-defined event (similarly as in Section 3.2.3.2, 70), which uses image processing techniques to detect a ball with the help of video cameras located in the forehead of Nao. This event has three input parameters: `robotIP`, `robotPort` and `checkingTime` have the same meanings that the corresponding variables `ip`, `port` and `interval` described before. Inside this event, when a ball is detected, we call the INI function `process_position` to process positions of the ball and the robot, and also specify the appropriate direction and velocity for the robot's movement. The position of the ball is attached to NAO's torso reference, and the calculation is done assuming an average ball size (diameter: 0.06 m) (see more details in [Ald13a]).

The event `@every` (lines 34-50) is applied to control the robot to move towards the target every 200 milliseconds. The INI function `reach_to_target` is used to determine a suitable velocity for the robot and to control the robot moving towards the ball. The robot only moves when all needed parameters related to orientation, velocity and frequency are specified. Each call of that function makes one walking step. During execution, the robot may adjust the direction and velocity to make them more well-suited since the ball may

change its position. As a result, after each step when the robot comes to the new location, we calculate the direction error. If the error for θ exceeds the allowed threshold (e.g. 10 degrees), the variable `needAdjustDirection` becomes `true` and some adjustments will be applied so that the robot walks in the correct way. We use the variable `i` to reset some parameters. When `i > 3`, this means that the robot already walked for three successful steps without checking again the position of the ball. In this case, by resetting some parameters to zeros, the robot will stop temporarily. Then it waits to detect the ball again to check whether during its displacement, the ball has moved to another place or not. If yes, Nao gets the updated position of the ball, then continues to walk and reach it.

In our program, we synchronize the two events `@ballDetection` and `@every` in order to avoid data access conflicts and unwanted behavior. For example, the robot is controlled to walk to the ball only when all needed parameters are calculated. Besides, we want to ensure that during the calculation of parameters, the robot is not moving so that the measured numbers are correct and stable. Consequently, we add the notation for synchronization, i.e. `$(...)` before each event (line 20 and line 34). Additionally, the event `@every` is also synchronized with itself so that each robot step does not overlap with others.

6.2.4 Testing Results, Evaluation and Future Work

When running in experiment, our program completed well the desired requirements. The robot detected an orange ball in the space and then followed it. When the ball was moved to another position, Nao also changed the direction and speed to reach the ball if needed. A demonstration video can be watched on YouTube [Le12].

We cannot apply our tool INICheck and SPIN to model-check the object-tracking program for interesting properties because some of the implementation is hidden in INI functions (written in Java) for controlling and interacting with Nao. If we can “extract" the underlying prototype for Java code and represent it in INI, we can solve the problem, although it is a non-trivial task. Besides, this limitation is also inherent in model checking in general since applying the technique requires us to “abstract” (simplify) a program first.

For future work, we will extend our example by adding more features to our program such as detecting and avoiding obstacles on the way to the target and control robot's hands

to catch the object. We also have a plan to develop more practical applications running on Nao. For example, we can build a program which may recognize human voice commands, and then control the robot to act according to the desired behavior.

Chapter 7

Conclusion and Future Work

Research on programming languages is considered as a central and fundamental topic in computer science that concerns all aspects related to programming. As software continuously evolves, programming languages also should evolve with an appropriate pace. Louden *et al.* [LL11] stated that with new advances in computer technologies, "there will be room for new languages and new ideas, and the study of programming languages will remain as fascinating and exciting as it is today". In the era of ubiquitous (pervasive) computing, to build high-quality applications and systems to comfort human life, developers still seek for better programming languages in terms of some major criteria like user-friendliness, robustness, effectiveness, customization and easy-to-validate. Arising trends like multithreading, intelligent computing, and smart environments also need to be taken into account.

In our thesis, we presented a new programming language called INI, which combines both event-based and rule-based styles. The introduction of event designing, formal techniques applied on context-aware and multithreaded programming, along with case studies are important contributions of this thesis. Our language can be used to write many kinds of applications, particularly those who need to take advantage of context-awareness and concurrency such as M2M systems, robots, embedded systems, interactive applications, monitoring and controlling systems, manufacturing systems, etc. Context information can be captured and handled through various kinds of events (built-in and user-defined ones). Events in INI may be executed in parallel either synchronously or asynchronously, depending on users' requirements and specifications. Furthermore, with the event reconfiguration

mechanism, programmers can write self-adaptive software more easily to adapt to variabilities in the execution environment. Along with events, rules may also be applied to react to changes when some conditions are satisfied. The major advantage of using INI is that it provides explicitly programmers with many advanced features while it keeps the unnecessary complexities behind. For that reason, INI programs require less effort and time to write and also are less error-prone. Additionally, INI comes with many formal features that makes INI programs more reliable. INI enjoys a strong type system so that all type conflicts are prohibited and unsound programs are excluded. The semantics of INI is also clearly defined (see Chapter 4) to allow programmers know deeply about the behavior of their programs. Besides, we also support programmers to take advantage of model checking techniques to verify their INI programs in order to detect possible specification errors at an early stage during the development life-cycle. Our tool INICheck can convert a non-trivial subset of INI to Promela, and then SPIN is applied to check the desired properties or required constraints. Programmers can additionally check which execution trace leads to the bugs (through the interactive mode) and get hints on how to fix the problems. Therefore, the system quality is enhanced and risks that may happen later are avoided. Briefly, besides general classical languages, programmers now have INI and INICheck as novel and well-defined tools to build and validate context-aware reactive applications, which may help them to reduce development and maintenance costs.

We applied our language in the scope of the MCUBE project, in which we developed a M2M gateway. The task of the M2M gateway is to capture and send data through a network with a schedule specified by the user. In our implementation, various kinds of data like image or sound from different sensors are collected concurrently and transmitted through the network at a given preferable time. The M2M gateway can be used in a factory to collect working condition (e.g. light, toxic elements, security, etc.), on a field to gather data about crop, or in a weather station to aggregate climate data (e.g. humidity, air quality, temperature, wind velocity, light, etc.). Furthermore, we built an INI object-tracking program running on the humanoid robot Nao. In this example, we control a robot to detect a small ball in the space and follow it. When the ball moves to another position, the robot automatically adjusts its direction and velocity in order to fulfill its task

properly. Object tracking and understanding can be applied for many purposes: security and surveillance in important places, gesture recognition, analysis of traffic-flow, detection of accidents, wild animals monitoring, etc. Those case studies justify the usefulness and efficiency of INI for programming context-aware and event-based embedded systems that become more and more popular in many activities.

When everything is taken into account, we see that INI satisfies the essential requirements for modern context-aware programming languages: an intuitive syntax, a well-defined semantics, a strong type system, a support for concurrency, an ability for dynamic behavior adaptation, and a support for automatic verification. These features allow programmers to build strong and flexible context-aware adaptive and reactive systems.

For the future work, we will improve the features of INI. Specifically, we will add more primitives (e.g. built-in events/functions, data structures). For instance, we will add a built-in `Date` type with + and − operators. Date literals will need to be specified, as well as date formatting. For type consistency, dates may be seen as durations from an initial date. In order to improve the reliability of INI programs, we can develop a fault-tolerance mechanism (a literature review on this technique is found in [Sah06]) to handle possible failures during adaptation execution. If a failure happens when our systems try to adapt to the context, it is essential to recover from that failure so that the system state is still stable. Furthermore, INI can be extended to support distributed systems [CDKB11]. Distributed event-based systems [MFP06, HB10] focus on establishing a manner for event-based components to work together through traditional notification (request/reply) mode. Additionally, to support programmers to manage, organize, write and maintain INI code easily, we will develop an Integrated Development Environment (IDE) for INI (based on the open-source IDE named Eclipse [The13]) with basic features like project and file management, syntax highlighters, syntax check, templates for developing user-defined events, debugging, and advanced features such as performance modeling and prediction (based on analytic approaches) and benchmarks.

Moreover, we have a plan to improve our tool INICheck so that it can translate a larger subset of INI to Promela, especially on how to model efficiently timing events. Formal

techniques will be used to prove that the semantics of an INI program and the translated Promela code are equivalent. Other state-of-the-art static analysis techniques such as code metrics [Pre10], partial evaluation [Zen08], data/control-flow analysis [KSK09, SS13], symbolic analysis [FS03] or abstract interpretation [Mas03] may also be applied to optimize and enhance the performance and quality of INI programs. Possible scenarios are using those techniques to detect programming mistakes such as the use of undefined variables, out of bounds variables, unused variables and/or code. Checking these anomalies helps us to eliminate potential subtle bugs.

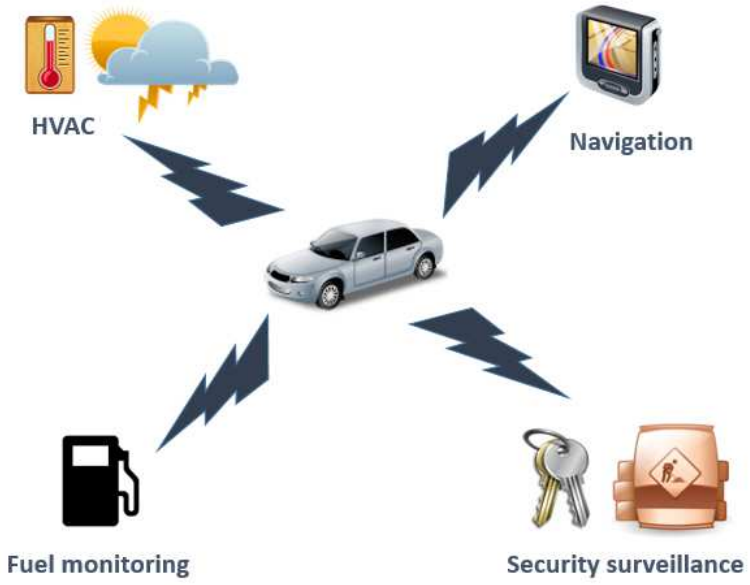


Figure 7.1: An intelligent automotive system.

Last but not least, we will apply INI to more cases, especially in daily life and industry. Candidate domains are using INI for controlling smart devices in a smart home, programming manufacturing robots or transportation systems. For instance, programmers may develop an intelligent automotive system as illustrated in Figure 7.1. In intelligent cars, the drivers do not need to bother too much about how to use and control properly and effectively their vehicles since everything (e.g. essence level, speed, weather, location, surrounding environment, etc.) can be automatically monitored and processed. Similarly, people can enjoy comfortable lives in smart homes, where all equipment (e.g. televisions, air conditioners, microwaves, refrigerators, etc.) know the way to please their owners,

especially for the children and the elderly. For example, a refrigerator stack may open automatically when someone approaches or a smart TV may recognize the changes of users' feelings to broadcast appropriate channels/movies. In a factory, we can develop a lighting control system that may adjust the artificial light sources depending on time and natural lighting condition to save power.

Résumé

Introduction

Les applications réactives et sensibles au contexte sont des applications intelligentes qui observent l'environnement (ou contexte) dans lequel elles s'exécutent. Elles adaptent, si nécessaire, leur comportement en cas de changements dans ce contexte, afin de satisfaire les besoins ou anticiper les intentions des utilisateurs. Les systèmes sensibles au contexte sont devenus l'un des concepts les plus passionnants de l'informatique ubiquitaire car ils couvrent un large spectre de domaines d'application: systèmes intelligents adaptatifs, systèmes de surveillance et de contrôle, robots, applications mobiles, systèmes de fabrication, programmes interactifs, etc [Dar09].

Les premiers travaux sur l'informatique sensible au contexte sont dus à Schilit *et al.* [SAW94]. Depuis lors, ces travaux ont été étendus dans de nombreuses directions. Cependant, l'écriture d'applications réactives sensibles au contexte reste toujours difficile et exige beaucoup de soin. Une des contraintes essentielles est qu'elles doivent être à la fois robustes et suffisamment souples pour être reconfigurées et s'adapter à des changements de contexte tels que: emplacement, connectivité, ressources, sécurité et préférences des utilisateurs. En d'autres termes, elles demandent un mécanisme bien défini pour gérer des sources très variées d'informations venant du contexte.

Les langages généralistes comme Java, C/C++, .Net ou Python peuvent être utilisés pour programmer de telles applications, mais l'augmentation de la demande a induit plusieurs travaux d'extension de ces langages pour faciliter leur développement. Citons notamment ContextJ pour Java [AHHM11] ou AwareC# pour C# [RS06]. Néanmoins, ces extensions restent peu pratiques pour les développeurs puisque les nouveaux concepts et

notations de l'extension sont mélangés avec ceux du langage de base. En d'autres termes, l'adaptation n'est pas aussi intuitive et simple qu'elle le pourrait. En conséquence, écrire et maintenir des applications sensibles au contexte demande encore beaucoup trop de temps et d'efforts.

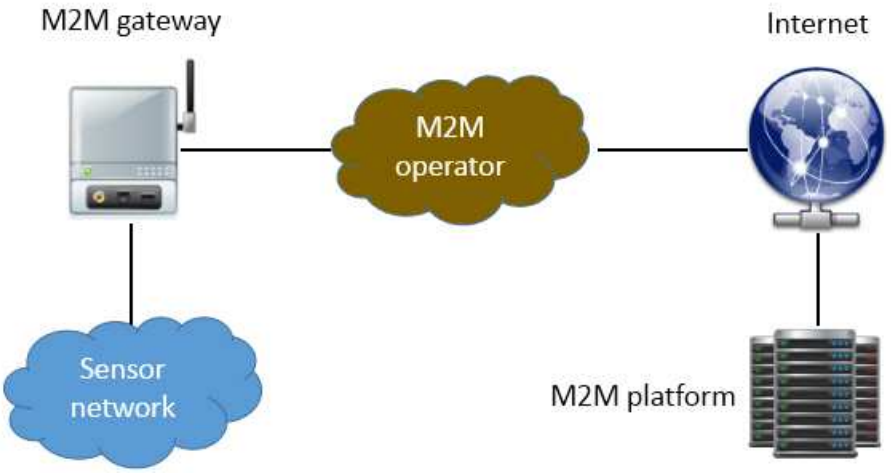
C'est pour permettre une écriture plus intuitive et directe que nous avons développé dans cette thèse un nouveau langage appelé INI. Pour observer les changements dans le contexte et y réagir, INI s'appuie sur deux paradigmes: la programmation événementielle et la programmation à base de règles. Événements et règles peuvent être définis en INI de manière indépendante ou en combinaison. En outre, les événements peuvent être reconfigurés dynamiquement au cours de l'exécution. Un autre avantage d'INI est qu'il supporte la concurrence afin de gérer plusieurs tâches en parallèle et ainsi améliorer les performances et la réactivité des programmes. Nous avons utilisé INI dans deux études de cas : une passerelle M2M multimédia et un programme de suivi d'objet pour le robot humanoïde Nao. Enfin, afin d'augmenter la fiabilité des programmes écrits en INI, un système de typage fort a été développé, et la sémantique opérationnelle d'INI a été entièrement définie. Nous avons en outre développé un outil appelé INICheck qui permet de convertir automatiquement un sous-ensemble d'INI vers Promela pour permettre un analyse par model checking à l'aide de l'interpréteur SPIN.

Ce travail s'intègre dans le cadre d'un projet FEDER : MCUBE (Multimedia 4 Machine 2 Machine) financé par le programme français "Compétitivité régionale et emploi 2011–2013". Ce projet vise à fournir un sytème (framework et langage de programmation) Machine-2-Machine (M2M) générique pour des applications multimédia, impliquant par exemple le recueil et l'analyse de sons et images (voir la figure ci-dessous).

Paradigmes de programmation

Programmation à base de règles

La programmation à base de règles existe depuis de nombreuses années et ses concepts fondamentaux sont présents dans un grand nombre de domaines de l'informatique. Une règle est constituée de deux parties: une prémisse (ou contrainte ou garde) et l'action



Architecture générale du projet MCUBE.

correspondante. Une règle peut être vue comme une instruction ou commande qui ne s'applique que dans certaines situations, elle ressemble en ce sens à la construction `if-then` [Hil03]. Cependant l'évaluation d'une règle, plus exactement de sa prémisse, ne se limite généralement pas à `true/false` ; elle se rapproche en de nombreux points du filtrage par motif (*pattern matching*) employé dans les langages fonctionnels.

La programmation à base de règles est souvent utilisée pour le développement d'applications intelligentes réactives, de systèmes experts ou de systèmes autonomes. Les règles sont également bien adaptées pour la programmation de systèmes sensibles au contexte, car elles permettent d'exprimer naturellement la dépendance d'une action au contexte. De nombreux langages à base de règles ont été développés, citons notamment CLIPS [CLI13] ou Jess [FH12]. Ceux-ci sont relativement proches les uns des autres, les variations, outre la syntaxe, se situent principalement au niveau de l'ordre d'évaluation des règles: il peut être séquentiel ou arbitraire, ce qui change le comportement du programme lorsque plusieurs règles sont applicables en même temps.

Programmation événementielle

La programmation événementielle est un paradigme de programmation où le flux d'exécution est déterminé par des événements, qui sont gérés par des gestionnaires d'événements ou *callbacks*. Un callback correspond aux instructions qui sont appelées lorsque l'événement

(c'est-à-dire quelque chose d'important) arrive [DZK+02]. Les événements sont généralement utilisés pour se déclencher périodiquement ou pour surveiller les changements qui se produisent dans l'environnement. Ainsi, toute forme de surveillance peut être considérée comme compatible avec la programmation événementielle [MFP06], et les exemples sont nombreux : surveiller le niveau d'énergie d'un système, la santé d'un patient, observer le comportement des utilisateurs, suivre la position d'un objet, etc.

Au cours des dernières années, la programmation événementielle s'est imposée comme méthode efficace pour l'interaction et la collaboration dans le cadre de l'informatique ubiquitaire. Ce paradigme de programmation demande généralement moins d'effort que la programmation classique et peut conduire à un logiciel plus robuste [DZK+02]. Il est particulièrement adapté à de nombreux types d'applications: applications M2M, systèmes embarqués, robotique, applications réactives sensibles au contexte, etc. En conséquence, plusieurs langages de programmation (ou extensions de langages) événementielle ont été développés récemment: EventJava [EJ09] est une extension de Java, tandis que EventScript [CK08] et UrbiScript [Gos11] sont des langages autonomes. Mais ces langages restent limités. Tout d'abord, les événements ne sont pas définis de manière intuitive et directe mais mélangés à d'autres concepts et éléments du langage. De plus, ces langages ne permettent généralement pas la synchronisation d'événements, un point essentiel lorsque plusieurs événements sont déclenchés simultanément. Enfin, ils ne permettent pas non plus de modifier dynamiquement le comportement des événements afin de s'adapter aux changements de l'environnement d'exécution. Autrement dit, un événement a toujours un comportement figé à la compilation, qui n'est pas forcément le plus adapté à toutes les situations données.

INI : Présentation générale

INI [1] est un langage de programmation développé à l'ISEP depuis 2010 [LP12] et dont l'interpréteur s'exécute dans une machine virtuelle Java (JVM). Cette JVM nous permet d'avoir un environnement de développement puissant et flexible, et adapté à des langages autres que Java [Gho10]. Bien qu'INI soit lié à la JVM, sa syntaxe et sa sémantique ne

[1] Le nom INI représente un événement (le N) passant entre deux interfaces (les I).

sont pas identiques à celles de Java.

Un programme INI est tout d'abord composé de fonctions. Ces dernières combinent événements et expressions logiques (utilisées pour spécifier des conditions d'activation) liées à une action (une liste d'instructions). Ainsi, la caractéristique principale d'INI est l'utilisation conjointe d'événements et de règles qui peuvent être définis soit indépendamment soit en combinaison. Les événements s'exécutent en parallèle de façon synchrone ou asynchrone, en fonction des contraintes, et il est possible de redéfinir leur comportement de manière dynamique pendant l'exécution. Il est également possible de développer des événements dans d'autres langages comme Java ou C/C++.

INI permet aussi des constructions plus usuelles, telles que les expressions arithmétiques/logiques, les dictionnaires, les listes et ensembles d'expressions ou l'instruction conditionnelle `case` (similaire à l'instruction `if`). Enfin, INI est un langage typé, dont les types sont inférés à la compilation.

Le code suivant est un exemple de fonction INI qui calcule la factorielle d'un nombre donné en argument. La fin d'une instruction est indiquée par un saut de ligne.

```
// Calcul de la factorielle de n avec INI
function fac(n) {
    @init() {
        f=1
        i=1
    }
    i <= n {
        f=f*i++
    }
    @end() {
        return f
    }
}
```

Les événements dans INI

Événements natifs INI permet de spécifier tous les types d'événements (déroulement du temps, changement d'état, phénomènes physiques). À chaque événement du corps de la fonction correspond un callback de gestion (ou *actions*), déclaré au même endroit. Ce callback est invoqué à chaque fois que l'événement se produit (l'exécution du callback représente l'instance de l'événement), par défaut de manière asynchrone. Syntaxiquement,

un événement en INI commence par `@`, immédiatement suivi de sa sorte, puis des paramètres d'entrée et de sortie correspondants. Enfin, vient le callback de l'événement (l'action), entre accolades `{ ... }`.

Les paramètres d'entrée sont des paramètres de configuration servant à ajuster le comportement, tandis que les paramètres de sortie sont le nom des variables auxquelles le callback affectera des valeurs. Ils peuvent être considérés comme les mesures associées à l'occurrence de l'événement. Ces variables, de même que toute variable INI, ont une portée lexicale qui s'étend à tout le corps de la fonction. Les deux types de paramètre sont optionnels. De plus, un événement peut aussi être lié à un identifiant, de façon à pouvoir y faire référence (e.g. à des fins de synchronisation ou de reconfiguration). Par exemple, l'événement `e:@sampleEvent[iParameter1 = v1](oParameter1,oParameter2)` est identifié par `e`, paramétré par `iParameter1` qui a pour valeur `v1`, et possède deux paramètres de sortie, qui seront affectés aux variables `oParameter1` and `oParameter2`, utilisables à n'importe quel endroit de la fonction, comme spécifié ci-dessus.

Pour permettre aux programmeurs d'écrire du code plus facilement, INI est nativement doté de plusieurs sortes d'événements communs: `@init()` (sert au tout début de la fonction, pour initialiser des variables) `@end()` (appelé lorsque plus aucun gestionnaire d'événement n'est en train d'être exécuté et que la fonction est sur le point de se terminer), `@every[time:Integer]()` (se déclenche périodiquement, comme spécifié par son entrée, en millisecondes), `@update[variable:T](oldValue:T, newValue:T)` (invoqué lorsque la valeur de la variable passée en paramètre d'entrée change) et `@cron[pattern:String]()` (planifie l'appel du callback, grâce à son paramètre d'entrée correspondant à un motif CRON).

Événements définis par l'utilisateur Les programmeurs peuvent aussi développer (en Java ou en C/C++) leurs propres sortes d'événement. Cette procédure, expliquée dans le corps de la thèse, ne sera pas abordée ici faute de place. Nous nous contenterons de discuter l'intégration des événements correspondants dans les programmes INI. C'est de cette manière qu'on traite par exemple les informations fournies par des capteurs. Pour illustrer notre propos, imaginons un programme qui doit récolter des informations provenant de cap-

teurs/appareils sondant un patient et mesurant des paramètres vitaux tels que température corporelle, pression sanguine, fréquence cardiaque, etc.

Dans notre programme INI, nous pouvons définir des événements séparés prenant en charge chacune de ces tâches. Par exemple, nous pouvons définir l'événement `@bloodPressureMonitoring` qui observera la pression sanguine avec une période définie par son paramètre d'entrée `bpPeriod` (en heures). De plus, son paramètre de sortie, `pressure`, indique la valeur courante de la mesure de pression. Dans le callback de l'événement de l'exemple ci-dessous, grâce à la construction `case` nous avons défini des comportements qui dépendent de la valeur de la pression, tels qu'alerter les infirmières/médecins si elle est trop importante. Les autres événements sont définis de manière similaire, et tous ceux-ci s'exécutent en parallèle de manière à pouvoir effectuer plusieurs tâches en même temps.

```
// Un programme INI de telesurveillance de sante
function main() {
    b:@bloodPressureMonitoring[bpPeriod = 2](pressure) {
        case {
            pressure > ... {...}
            default {...}
        }
    }
    t:@temperatureMonitoring[tempPeriod = 1](temperature) {...}
}
```

Synchronisation et reconfiguration des événements Par défaut, et à l'exception des événements `@init` et `@end`, les événements INI sont exécutés de manière asynchrone (c'est-à-dire toutes les instances peuvent s'exécuter en concurrence). Cependant, certains scénarios d'exécution peuvent demander à un événement `e0` d'être synchronisé sur d'autres événements `e1, ..., eN`. Cela veut dire qu'avant de pouvoir s'exécuter, l'instance de l'événement synchronisant `e0` doit attendre que tous les threads correspondant à des instances des événements cibles soient terminés. Le mécanisme de synchronisation devient nécessaire lorsque, par exemple, l'action de `e0` interfère avec l'action d'autres événements.

Notons aussi que les événements cibles peuvent aussi être synchronisés avec `e0`. Une synchronisation croisée est synonyme d'exclusion mutuelle: deux instances de deux événements mutuellement synchronisés ne peuvent pas être exécutés en même temps.

Les changements de l'environnement peuvent de plus être gérés à travers le mécanisme

de reconfiguration d'événements. Essentiellement, la reconfiguration d'événements consiste à modifier la valeur des paramètres d'entrée d'un événement donné. Pour cela, il faut appeler la primitive `reconfigure_event(eventId, [inputParam1 = value1, inputParam2 = value2,...])`. Les programmeurs sont aussi autorisés à arrêter et redémarrer les événements avec les primitives `stop_event([eventId1, eventId2, ...])` et `restart_event( [eventId1,eventId2, ...])`. Typiquement, un arrêt est recommandé avant une reconfiguration.

Les règles dans INI

Une règle dans INI se définit par la donnée d'une expression logique (la garde) et de l'action correspondante. Quand l'expression logique de la règle est évaluée à *true*, l'action est déclenchée.

Considérons de nouveau notre système de surveillance santé.

```
1  function main() {
2    ...
3    @bloodPressureMonitor[bpPeriod = 2](pressure) {...}
4    @temperatureMonitor[tempPeriod = 1](temperature) {...}
5    temperature > T0 && pressure < P0 {
6      // Action d'urgence
7    }
8  }
```

La règle des lignes 5–7 est déclenchée lorsque la température du patient et sa pression sanguine atteignent les seuils donnés.

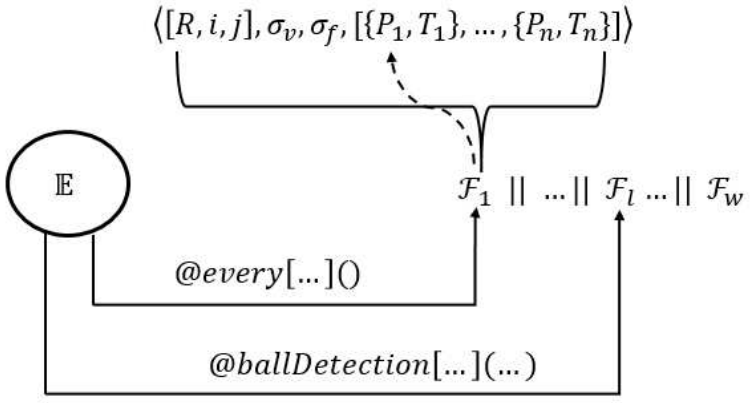
Sémantique

Notations et symboles

Comme mentionné plus haut, chaque fonction en INI contient des règles et des événements, ces derniers s'exécutant en parallèle dans l'environnement même de la fonction. Cela a aussi pour conséquence que les appels de fonctions seront évalués en parallèle (puisqu'elles peuvent être appelées par des événements). Nous reflétons cette structure complexe dans la sémantique, en la dotant de deux niveaux: au niveau supérieur nous trouvons $\overrightarrow{\mathcal{F}}$, le contexte d'exécution global. $\overrightarrow{\mathcal{F}}$ est composé de plusieurs contextes d'exécution de fonctions

(décrits ci-dessous) qui sont évalués en parallèle, et notés $\mathcal{F}_1 \,||\, \mathcal{F}_2 \,||\, ... \,||\, \mathcal{F}_w$. Chacun de ces contextes évalue *un* appel de fonction.

Nous adjoignons à ces w contextes d'évaluation de fonction un générateur d'instances d'événements $\mathbb{E}$. Celui-ci peut être considéré comme une boîte noire se caractérisant par son interaction avec $\overrightarrow{\mathcal{F}}$: $\mathbb{E}$ est en charge de récolter les paramètres d'entrée lors de l'enregistrement ou de la reconfiguration d'un événement, de le lier à un contexte de fonction particulier (celui dans lequel est défini l'événement). Il est aussi chargé de déclencher l'événement lorsque les conditions requises (dépendant à la fois des paramètres d'entrée et de sortie de l'événement) sont satisfaites. Enfin, lorsque l'événement a été déclenché et est exécutable (par exemple pas de conflit de synchronisation, ou d'événement non terminé, voir ci-dessous), $\mathbb{E}$ calcule la valeur des paramètres de sortie, les lie aux variables, et envoie le callback gestionnaire de l'événement dans le pool de threads adéquat du contexte de fonction F_l auquel l'événement à été lié lors de son enregistrement.
Ces éléments sont représentés dans la figure ci-dessous.



Contexte d'exécution et mécanisme de génération d'événements dans INI.

Les contextes d'évaluation de fonction F_l sont composés de:

- $[R, i, j]$, qui sert à évaluer les règles. `i` est l'indice de la règle dont la condition de garde doit être examinée, tandis que `j` compte le nombre de règles évaluées comme non applicables (condition de garde fausse) depuis le dernier succès.

- Une instruction remplace $[R, i, j]$ lorsque le corps d'une règle est en cours d'évaluation.

- σ_v^l est l'environnement des variables et des paramètres de la fonction. C'est une table d'associations $VariableName \rightarrow Value$. Il comprend les variables locales à la fonction ainsi que ses paramètres.

- σ_f^l est l'environnement de fonctions, c'est une table d'association: $FunctionName \rightarrow \Psi$, où Ψ représente l'ensemble des triplets d'abstraction de fonction: un environnement pour les variables σ_v, la liste R des règles du corps de la fonction, et la liste $\overrightarrow{\mathcal{C}}$ des événements (voir ci-dessous). Notons que σ_f^l est en réalité global, car il ne dépend pas de F_l (absence de création dynamique de clôtures, par exemple).

- un couple $\{P_k^l, T_k^l\}$ pour chacun des événements du corps de la fonction. P_k^l est le pool qui contient tous les threads correspondants aux instances de e_k déclenchées par $\mathbb{E}$ dont l'exécution n'est pas terminée. T_k^l est un booléen indiquant e_k est terminé. C'est un état qui est atteint après appel à `stop_event`, et dans ce cas, aucun nouveau thread ne peut être ajouté à P_k^l ; les anciens seront autorisés à terminer. Dans chaque thread, les instructions sont évaluées de la même manière que celles du corps des règles (le corps des événements et des règles ne contenant que des instructions). Nous notons Θ_P^l l'ensemble des pools de threads d'une fonction.

Nous définissions $\Sigma = [\sigma_v, \sigma_f, \Theta_P]$ et l'appelons aussi, par abus de langage, le contexte d'exécution de fonction, ou plus simplement le contexte. Ainsi, σ_f est l'ensemble des contextes abstraits d'exécution de fonction du programme.

L'évaluation d'une expression `expr` sera donc notée $\langle expr, \Sigma \rangle$. Par souci de simplicité, nous supposerons que $\overrightarrow{\mathcal{F}}$ contient un seul contexte de fonction, et nous laisserons $\mathbb{E}$ implicite lorsqu'il n'interagit pas avec $\overrightarrow{\mathcal{F}}$, ce qui est le cas la plupart du temps.

Enfin, notre sémantique est à la fois une sémantique à grand pas, auquel cas $\Downarrow$ est utilisé et indique que nous évaluons un contexte vers une valeur, et une sémantique à petit pas, auquel cas $\Rightarrow$ est utilisé et indique une transition entre états.

Sémantique des règles

Chaque règle comporte une condition de garde *le* qui est une expression logique associée à une action (le corps, une liste d'instructions). Lorsque *le* est évalué à `true`, l'action est

évaluée. Les règles sont testées séquentiellement et en boucle, en commençant par la première, jusqu'à ce que plus aucune règle ne soit applicable. En appelant r le nombre total de règles, la sémantique est donc la suivante:

- L'évaluation commence par la première règle:

$$\frac{}{\langle \triangleright,\ \Sigma\rangle\ \Rightarrow\ \langle [R,\ 0,\ 0],\ \Sigma\rangle}$$

  où "$\triangleright$" indique le début de l'évaluation du corps de la fonction.

- Aucune action n'est effectuée lorsque la condition de la règle s'évalue en `false` ; on passe directement à la règle suivante:

$$\frac{\langle le_i,\ \Sigma\rangle\ \Downarrow\ \langle false,\ \Sigma'\rangle}{\langle [R,\ i,\ j],\ \Sigma\rangle\ \Rightarrow\ \langle [R,\ (i+1) \bmod r,\ j+1],\ \Sigma'\rangle}$$

  où "mod" représente l'opération "modulo" qui donne le reste de la division euclidienne.

- L'action est exécutée lorsque la condition de la règle s'évalue en A `true` ; on passe ensuite à la règle suivante:

$$\frac{\langle le_i,\ \Sigma\rangle\ \Downarrow\ \langle true,\ \Sigma'\rangle \quad \langle body_i,\ \Sigma'\rangle\ \Rightarrow\ \langle \epsilon,\ \Sigma''\rangle}{\langle [R,\ i,\ j],\ \Sigma\rangle\ \Rightarrow\ \langle [R,\ (i+1) \bmod r,\ 0],\ \Sigma''\rangle}$$

  où ϵ représente l'intruction vide.

Sémantique des événements

Chaque instance d'événement e_{inst} a pour finalité de s'exécuter dans le pool de threads P correspondant. Les instructions de l'action EA correspondante (voir ci-dessous) y seront évaluées de la même manière que l'action des règles. Notons aussi que plusieurs instances du même événement peuvent être évaluées en même temps. Une instance d'événement est composée des parties suivantes:

- L'identifiant explicite (externe) de l'événement, *eid*, qui apparaît dans le programme INI (s'il est défini), et auquel les programmeurs peuvent faire référence (pour agir sur l'événement à travers les trois fonctions `stop_event`, `reconfigure_event` ou `restart_event`). De plus, chaque événement possède aussi un identifiant implicite

(interne), servant principalement à gérer les pools de threads. L'utilisateur ne peut y accéder, et il est défini seulement en l'absence d'identifiant externe (événement anonyme). Ainsi, un événement a toujours un identifiant soit externe, soit interne, mais jamais les deux en même temps. Nous assimilerons donc ces deux identifiants à *eid*.

- La sorte de l'événement (e.g. `@every`, `@ballDetection`, etc.).
- La condition de garde le, si elle existe.
- La liste L de tous les identifiants des événements sur lesquels e_{inst} est synchronisé.
- Les paramètres: l'ensemble des noms des paramètres d'entrée $IP = ip_1, ip_2, \cdots, ip_p$ et les valeurs correspondantes $iv_1, iv_2, \cdots, iv_p$; l'ensemble des noms des paramètres de sortie $OP = op_1, op_2, \cdots, op_q$ et les valeurs correspondantes $ov_1, ov_2, \cdots, ov_q$.
- L'action EA (qui n'est pas encore en cours d'exécution), qui est une séquence d'instructions. C'est le callback gestionnaire de l'événement, et c'est lui qui se chargera de réagir à l'occurrence de l'événement.
- La valeur booléenne $T_{e_{inst}.eid}$, qui indique si l'événement est terminé ou non. Par défaut $T_{e_{inst}.eid}$ vaut `false`. Un appel à `stop_event` ou à `restart_event` sur $e_{inst}.eid$ change la valeur de $T_{inst.eid}$ en `true` ou `false`, respectivement.

Toutes ces composantes, à l'exception de $T_{e_{inst}.eid}$ forment le callback $\mathcal{C}_{e_{inst}.eid}$.

Nous notons RE l'ensemble des instances d'événements exécutables d'un programme INI. Par exemple, si une balle est détectée par la caméra, l'événement `@ballDetection` génère une instance exécutable. RE fait partie intégrante de $\mathbb{E}$.

Comme il a été dit plus haut, $\mathbb{E}$ est le générateur d'instances d'événements, et possède les caractéristiques suivantes:

- $\mathbb{E}$ prend en charge l'enregistrement des événements et la génération des instances correspondants lors de l'occurrence de l'événement "physique". Il place cette instance dans RE (cf. point suivant).

- $\mathbb{E}$ contient donc l'ensemble des instances exécutables RE qui correspond aux événements qui ont physiquement eu lieu, mais dont l'action n'est pas encore en cours d'évaluation. En particulier, les instances d'événements de RE seront supprimées de RE seulement après que l'action EA aura été envoyée au pool de threads correspondant, ou bien si la condition de garde *le* s'évalue à `false`, ou encore si l'événement est terminé.

Le générateur d'événements $\mathbb{E}$ interagit avec le reste du programme INI à travers trois mécanismes: a) les appels de fonction, qui enregistrent auprès de $\mathbb{E}$ les événements déclarés dans le corps de celle-ci. De manière duale, lorsqu'une fonction est terminée, les événements correspondant sont désenregistrés ; b) le déclenchement d'événements et c) l'arrêt, la reconfiguration et le redémarrage d'événements. (a) et (c) sont des actions du contexte $\overrightarrow{\mathcal{F}}$ sur $\mathbb{E}$, tandis que (b) s'effectue dans le sens inverse.

Il y a trois cas possibles lors du traitement de e_{inst}:

- L'événement $e_{inst}.eid$ est terminé. Dans ce cas, l'instance ne doit pas être exécutée et elle est retirée de RE.

$$\frac{e_{inst} \in RE \quad T_{e_{inst}.eid} = true}{\mathbb{E} \ \langle cur,\ \Sigma\rangle \ \Rightarrow \ \mathbb{E}' \ \langle cur,\ \Sigma\rangle}$$

  où *cur* représente soit l'expression soit la règle en cours d'évaluation.

- au moins l'un des pools de threads des événements sur lesquels e_{inst} est synchronisé n'est pas vide. Dans ce cas, e_{inst} ne doit pas être exécuté immédiatement mais doit être conservée. Autrement dit, $\mathbb{E}$ doit attendre que tous les pools soient vides.

$$\frac{e_{inst} \in RE \quad \exists k \in e_{inst}.L \quad P_k \neq \emptyset}{\mathbb{E} \ \langle cur,\ \Sigma\rangle \ \Rightarrow \ \mathbb{E} \ \langle cur,\ \Sigma\rangle}$$

- autrement, e_{inst} est exécutée.

$$\frac{e_{inst} \in RE \quad T_{e_{inst}.eid} = false \quad \forall k \in e_{inst}.L \quad P_k = \emptyset}{\langle cur,\ \Sigma\rangle \ \Rightarrow \ \langle cur,\ \Sigma'\rangle}$$

  avec $\Sigma' = [\sigma'_v,\ \sigma'_f,\ [P'_1,\ T'_1, ..., P_{e_{inst}.eid} \ \cup \ Thread(e_{inst}.EA),\ T'_{eid}, ...]]$ et où $Thread$ $(e_{inst}.EA)$ est un nouveau thread créé pour exécuter l'action de l'événement.

Analyse statique

Système de types

Types natifs et types définis par l'utilisateur INI possède 5 types natifs pour les nombres (*Double*, *Float*, *Long*, *Int*, et *Byte*), un type *Char* et un type *Boolean*. De plus, INI est doté d'un type dictionnaire polymorphique: $Map(K, V)$, où K est le type des clés, et V le type des valeurs. Les listes sont des instances de ce type dictionnaire, avec $K = Int$ (en réalité, il s'agit donc d'un ensemble indexé) et sont aussi notées T^* (sucre syntaxique). Le type *String* est simplement $Char^*$.

L'ensemble des types numériques de INI est ordonné par la relation $\succ$. En particulier, cet ordre doit interdir d'assigner des nombres plus génériques à des nombres moins génériques: $Double \succ Float \succ Long \succ Int \succ Byte$.

Au delà de ces type natifs, les programmeurs peuvent définir leurs propres types en utilisant le mot clé *type*, suivi du nom du type commençant par une majuscule. Ainsi, il est possible de définir un type *Person* en écrivant: `type Person = [name:String, age:Int]`.

Inférence de types INI utilise l'*inférence de type* ce qui permet de définir les types de manière implicite. Ainsi, l'instruction *i=0* associe à i le type *Int*. Si la variable i est utilisée avec un autre type, par exemple avec *i=0.1* qui associe le type *Float* à i, une erreur de typage est déclenchée. De façon similaire, l'utilisation des crochets [] définira automatique une variable de type *Map*: l'expression *l[i]* indique à INI que l est de type *Map(Int,T)* (i.e. une liste dont les éléments sont de type T, où T représente n'importe quel type).

La plupart des types d'INI sont calculés par le moteur d'inférence qui s'appuie sur l'algorithme d'unification de Herbrand, tel que décrit par Robinson dans [Rob65]. Ce moteur prend en compte les fonctions polymorphiques, les types de données abstraits (ou types algébriques) et le sous-typage interne lié à $\succ$ pour les types numériques. Pour en savoir plus sur le typage dans INI, se reporter à la documentation de référence [LP12].

Vérification de programmes INI par *model checking*

Afin d'aider à détecter des erreurs dans un programme INI, nous avons développé l'outil `INICheck`, qui convertit automatiquement un sous-ensemble d'INI en Promela, le langage utilisé par le model checker SPIN. Promela est un langage de modélisation et pas d'exécution, toutes les fonctionnalités d'INI ne peuvent donc pas être converties. `INICheck` génère le code Promela à partir de l'AST (Abstract Syntax Tree) d'un programme INI. Plus spécifiquement, l'AST est parcouru et chaque construction est traduite vers la structure Promela correspondante. Notre outil possède les caractéristiques suivantes:

- l'ensemble des règles et des événements natifs peut être traduit,
- la structure des actions associées aux événements définis par l'utilisateur est conservée,
- le mécanisme de synchronisation des événements est converti.

`INICheck` peut être utilisé pour plusieurs buts : détection de boucle infinie, de code inatteignable, ou encore vérification de propriétés ou de contraintes exprimées en LTL (logique temporelle linéaire). La figure ci-dessous présente une vue d'ensemble de notre approche.

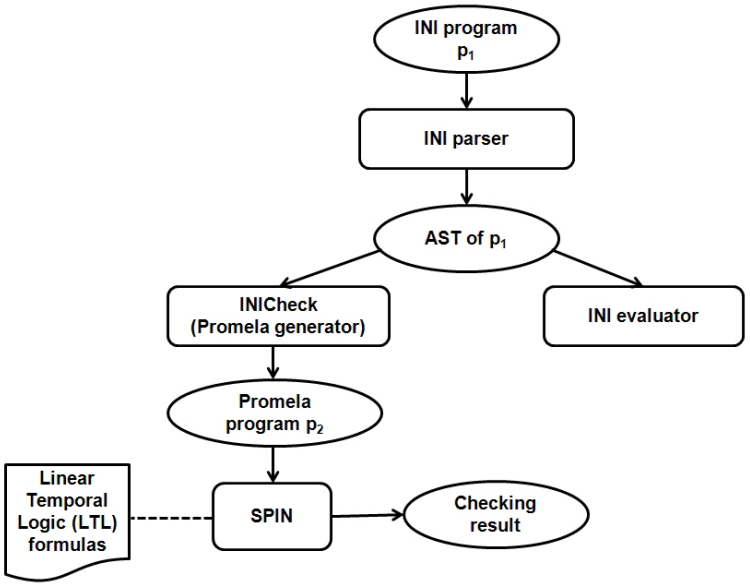
Études de cas

Une passerelle M2M multimédia

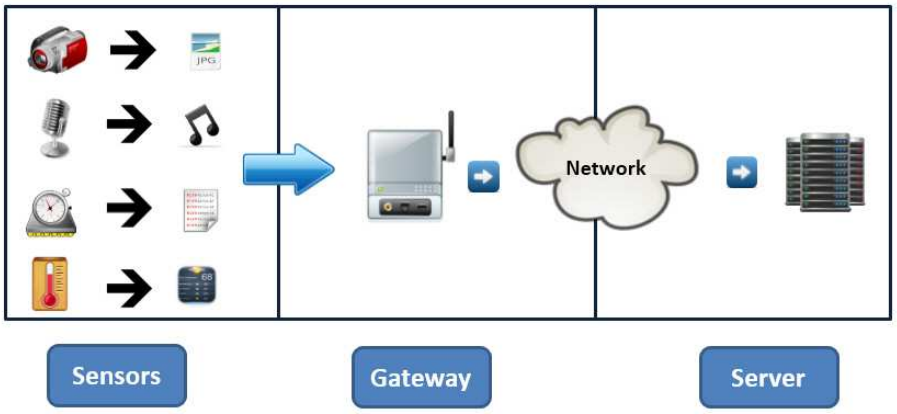
Nous avons utilisé INI dans le cadre des systèmes M2M (Machine to Machine), et plus précisément pour une passerelle multimédia dont le comportement est représenté ci-dessous.

Le programme que nous avons développé fonctionne en deux étapes:

1. Collecte de manière régulière des données multimédia (images, sons, etc.) obtenues par des capteurs ou des périphériques. Ces opérations sont gérées par les événements e1 et e2 du programme ci-dessous.



Vue d'ensemble du model checking d'un programme INI.



Comportement de la passerelle multimédia.

2. Transmission de ces données au serveur via le réseau à des fins de stockage et autres traitements coûteux en ressources. Cette transmission est gérée par l'événement @cron.

```
// Une passerelle M2M en INI
function main() {
    @init() {
        dataFolder = file("data")
        case {
            !file_exists(dataFolder) { mkdirs(dataFolder) }
        }
        zipFile = file("data.zip")
```

```
        keepParentFolder = true
    }
    // Deux evenements capturant des images et du son
    e1:@every[time = 60000]() {
        exec("gphoto2 --capture-image-and-download --filename "
         + "data/img" + time() + ".jpg")
    }
    e2:@every[time = 30000]() {
        exec("arecord -d 30 -f cd " + "data/sound" + time()
         + ".wav")
    }
    // Planification du telechargement FTP des donnees
    @cron[pattern = "0 09-18 * * 1-5"]() {
        stop_event([e1,e2])
        zip(dataFolder,zipFile)
        upload_ftp("server_address", "user_name",
         "password", zipFile, to_string(time()) + "data.zip")
        delete_file(zipFile)
        delete_folder(dataFolder, keepParentFolder)
        restart_event([e1,e2])
    }
}
```

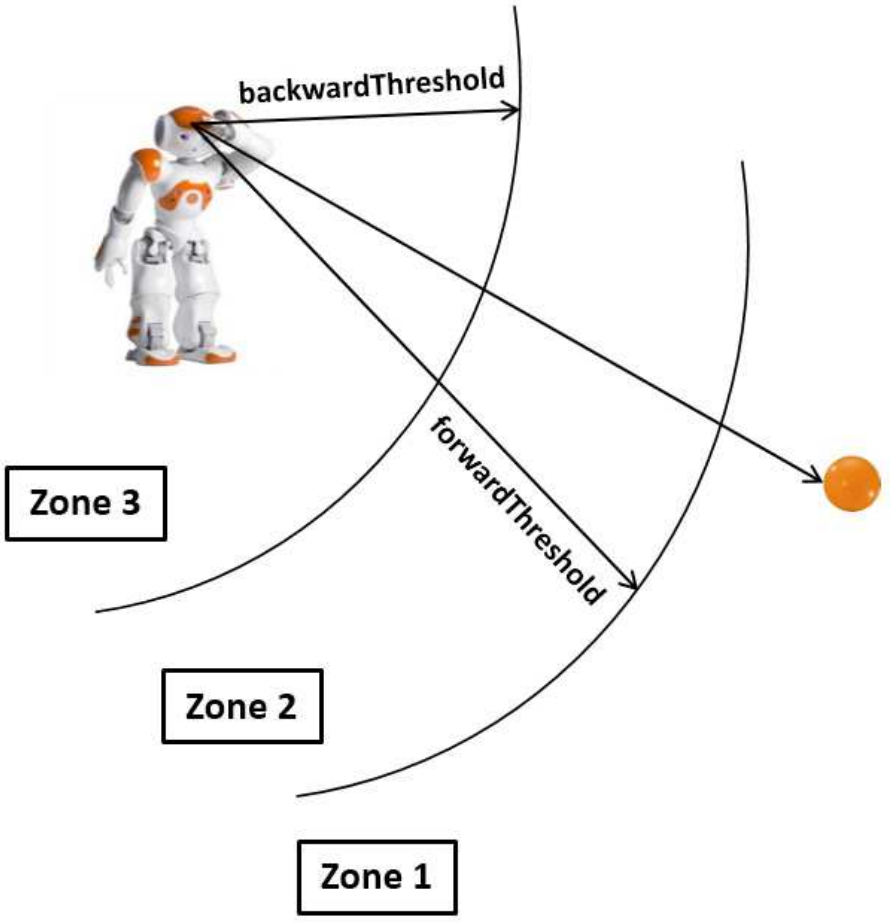
Suivre un objet avec NAO

La figure ci-dessous présente les positions relatives du robot NAO et de la balle qu'il doit suivre. Dans la zone 1, NAO est loin de la balle et doit s'en rapprocher ; dans la zone 3, NAO est trop proche de la balle et doit reculer ; enfin, dans la zone 2, NAO est à bonne distance et peut interagir avec la balle.

La figure ci-dessous présente le comportement général de notre programme.

Nous avons également utilisé INI pour contrôler un robot humanoïde de type NAO. Ce programme permet au robot de détecter une balle (événement b), puis de se diriger vers elle (événement e). Pendant l'exécution du programme, la balle peut être déplacée, et le robot modifie alors la direction et la vitesse de son déplacement.

```
function main() {
    // Initialisation
    @init() {
        forwardThreshold = 0.5
        backwardThreshold = 0.3
        interval = 1000
        stepFrequency = 0.0
        defaultStepFrequency = 1.0
        ip = "nao.local"
```



Positions relatives du robot NAO et de la balle à suivre.

```
10          port = 9559
11          useSensors = false
12          targetTheta = 0.0
13          robotPosition = [0.0,0.0,0.0]
14          stepX = 0.0
15          needAdjustDirection = false
16          i = 0
17      }
18
19      // Detection dans l'espace d'une balle
20      $(e) b:ballDetection[robotIP = ip, robotPort = port,
21       checkingTime = interval](ballPosition){
22          // Calcul des parametres, retournes dans un tableau
23          parameters = process_position(ip, port, ballPosition,
24           forwardThreshold, backwardThreshold, useSensors)
25          targetTheta = parameters[0]
26          robotPosition = parameters[1]
```

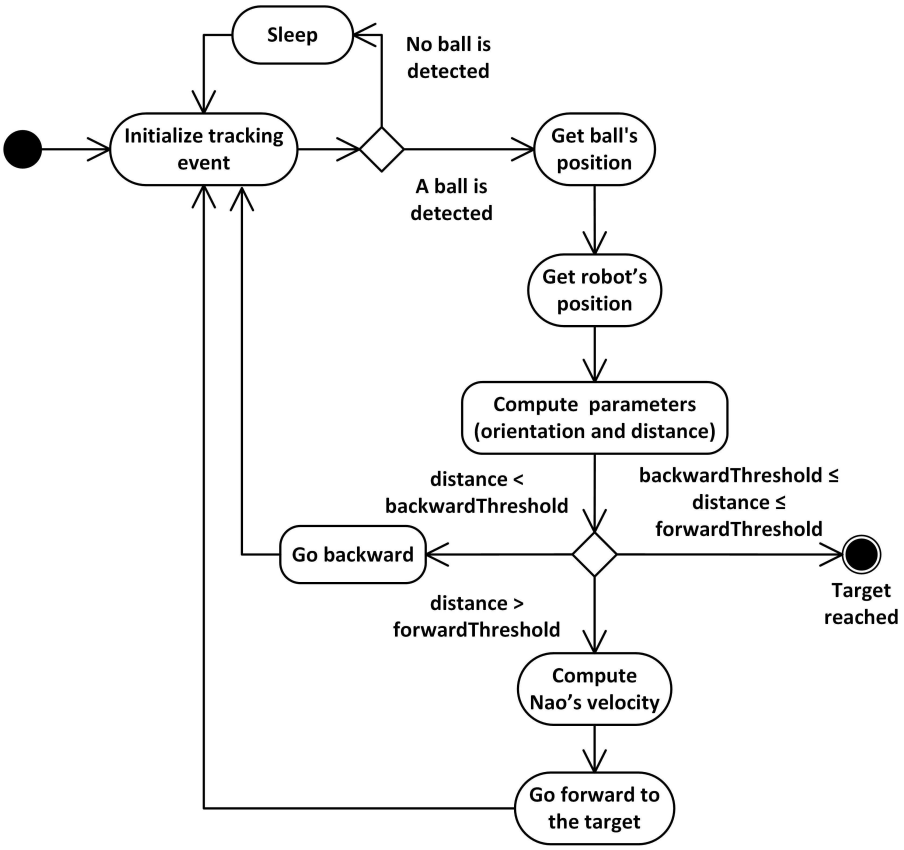


Diagramme d'activité du programme de suivi de balle.

```
27          stepX = parameters[2]
28          i = 0
29          needAdjustDirection = true
30          stepFrequency = defaultStepFrequency
31      }
32
33      // Controle periodique du robot
34      $(b,e) e:@every[time = 200]() {
35          // avancer de un pas si une balle est detectee
36          needAdjustDirection = reach_to_target(ip, port,
37           stepFrequency, robotPosition, stepX, targetTheta,
38          needAdjustDirection, useSensors)
39          i++
40          case {
41              // Remise a zero des parametres apres
42              // trois pas consecutifs
43              i>3 {
44                  stepX = 0.0
45                  targetTheta = 0.0
46                  stepFrequency = 0.0
```

```
47                  }
48              }
49          }
50  }
```

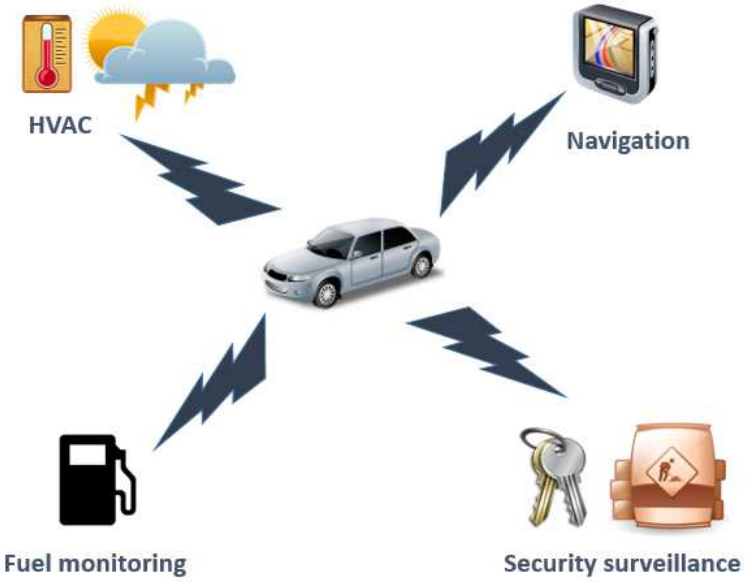
Conclusion et travaux futurs

Cette thèse a permis de développer un nouveau langage de programmation appelé INI pour les applications réactives et sensibles au contexte. INI s'inspire des paradigmes de programmation événementielle et à base de règles mais étend les approches déjà proposées, notamment avec un mécanisme de reconfiguration répondant au besoin d'adaptation dynamique à l'environnement. Les événements s'exécutent d'autre part en parallèle (de façon asynchrone ou synchrone) afin d'accélérer l'exécution et d'améliorer les performances du programme. Un ensemble d'événements prédéfini est proposé, mais il est également possible d'écrire ses propres événements en Java ou C/C++. L'apport de la programmation à base de règles est essentiel car elle permet de définir simplement les conditions à satisfaire pour déclencher les actions associées.

Une des contributions les plus importantes de la thèse a été d'appliquer des méthodes formelles à INI, langage utilisant massivement les threads. Un tel travail donne clairement un avantage qualitatif à INI dans le domaine de la programmation sensible au contexte, dans celui de la programmation par événements et, dans une moindre mesure, dans celui de la programmation par threads. Ainsi, nous avons entièrement défini la sémantique opérationnelle du langage. Son originalité est d'être construite à deux niveaux, et d'avoir combiné une sémantique à petits et à grands pas. Cette formalisation très précise de l'évaluation d'un programme peut en particulier être utilisée comme une référence par les programmeurs, mais aussi à des fins de spécification ou de vérification. Nous avons également développé INICheck, un outil permettant de traduire automatiquement un programme INI en Promela afin de vérifier ce programme avec le model checker SPIN. Cet outil permet ainsi d'améliorer la confiance du programmeur sur le code et son comportement.

Enfin, INI a été utilisé dans deux cas d'application réels: une passerelle multimédia M2M et un programme de suivi d'objet fonctionnant sur le robot humanoïde Nao.

Dans le futur, nous envisageons d'étendre INI pour supporter les systèmes distribués, une des difficultés étant alors de définir comment les événements pourraient agir entre eux via, par exemple, le mode traditionnel de notifications (request/reply). Par ailleurs, nous pensons développer un mécanisme de récupération du contexte de sorte que lorsqu'une reconfiguration échoue, le système revienne à l'état stable précédent. INICheck doit être amélioré afin de convertir un sous-ensemble plus grand de INI vers Promela, la question se pose en particulier sur la façon de modéliser efficacement les événements et leur synchronisation. Il est également nécessaire de prouver notre conversion, autrement dit de montrer que le programme INI et sa traduction en Promela ont des sémantiques équivalentes. Il est aussi possible d'améliorer, ou d'augmenter, les analyses statiques et dynamiques qui sont réalisées afin d'accroître la fiabilité des programmes INI. Finalement, nous réfléchissons à des applications plus concrètes pour évaluer au mieux les capacités d'INI (par exemple, pour la fabrication de programme ou les systèmes automobiles).



Un système automobile intelligent.

Mots-clefs: Programmation événementielle, Programmation à base de règles, Applications sensibles au contexte, Smart Computing, Programmation de robots, Programmation concurrente, Programmation embarquée, Vérification et validation, Analyse statique,

Model checking.

Bibliography

[Abd01] Slim Abdennadher. Rule-based constraint programming: Theory and practice, 2001. 55

[AHH+09] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, COP '09, pages 6:1–6:6, New York, NY, USA, 2009. ACM. 39

[AHHM11] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Computer Software*, 28(1):272–292, 2011. 39, 177

[AHM+10] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Proceedings of the 9th international conference on Software composition*, SC'10, pages 50–65, Berlin, Heidelberg, 2010. Springer-Verlag. 39

[Ald13a] Aldebaran Robotics. Nao software documentation. `http://www.aldebaran-robotics.com/documentation/`, 2013. 163, 166, 167, 168

[Ald13b] Aldebaran Robotics. Nao's homepage. `http://www.aldebaran-robotics.com`, 2013. 19, 163, 164

[All12] TinyOS Alliance. TinyOS homepage. `http://www.tinyos.net/`, 2012. 38

[Alm11] J.B. Almeida. *Rigorous software development: An introduction to program verification*. Undergraduate Topics in Computer Science. Springer London, Limited, 2011. 22

[Als12] Alsa group. Alsa. `http://www.alsa-project.org/`, 2012. 155

[Ama12] Lucas Amador. *Drools developer's cookbook*. Packt Publishing, January 2012. 55

[Amz10] Amzi! Inc. Expert systems in Prolog. `http://www.amzi.com/ExpertSystemsInProlog/xsiptop.php`, 2010. 59

[Aud13] Samuel Audet. JavaCPP. `http://code.google.com/p/javacpp/`, 2013. 73

[Auy06] Tak Auyeung. Robot programming in C. `http://www.drtak.org/teaches/ARC/cisp299_bot/book/book.pdf`, 2006. 162

[BA08] Mordechai Ben-Ari. *Principles of the SPIN Model Checker*. Springer-Verlag London, 2008. 131, 133, 135, 136, 138, 142, 147

[BA10] Mordechai (Moti) Ben-Ari. A primer on model checking. *ACM Inroads*, 1:40–47, March 2010. 130

[Bai05] Jean-Christophe Baillie. URBI: Towards a universal robotic low-level programming language. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'05)*, pages 820–825, 2005. 50

[Bak06] Steffen Van Bakel. Applied operational semantics, 2006. 96

[Bar05] Jakob E. Bardram. The Java context awareness framework (JCAF) - A service infrastructure and programming framework for context-aware applications. In *Proceedings of the Third international conference on*

Pervasive Computing, PERVASIVE'05, pages 98–115, Berlin, Heidelberg, 2005. Springer-Verlag. 39

[BBF+10] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: Model-checking techniques and tools.* Springer Publishing Company, Incorporated, 1st edition, 2010. 128

[BBF+12] Bernard Berthomieu, Jean-Paul Bodeveix, Mamoun Filali, Hubert Garavel, Frederic Lang, Didier Le Botlan, François Vernadat, and Silvano Dal Zilio. The syntax and semantics of Fiacre – Version 3.0, 2012. 97

[BCC+07] Antonio Brogi, Javier Cámara, Carlos Canal, Javier Cubo, and Ernesto Pimentel. Dynamic contextual adaptation. *Electron. Notes Theor. Comput. Sci.*, 175(2):81–95, June 2007. 38

[BCHM09] Y. Bar-Cohen, D. Hanson, and A. Marom. *The coming robot revolution: Expectations and fears about emerging intelligent, humanlike machines.* Springer-Verlag New York, 2009. 161, 163

[BD04] Kim B. Bruce and Andrea Danyluk. Event-driven programming facilitates learning standard programming concepts. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 96–100, New York, NY, USA, 2004. ACM. 40

[BDHN10] Jean-Christophe Baillie, Akim Demaille, Quentin Hocquet, and Matthieu Nottale. Events! (reactivity in Urbiscript). *CoRR*, abs/1010.5694, 2010. 50

[BEH12] D. Boswarthick, O. Elloumi, and O. Hersent. *M2M communications: A systems approach.* Wiley, 2012. 28

[Bek08] G.A. Bekey. *Robotics: State of the art and future challenges.* World Scientific, 2008. 161

[BJSS09] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *ASE*, pages 161–169, 2009. 148

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* The MIT Press, 2008. 128, 130, 138

[BMZ+05] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17(5):309–332, September 2005. 35

[Boa95a] ESA Board. Guide to software quality assurance, 1995. 128

[Boa95b] ESA Board. Guide to software verification and validation, 1995. 128

[Boe07] Barry Boehm. EQUITY keynote address, March 2007. 21

[Bou13] J.L. Boulanger. *Static analysis of software: The abstract interpretation.* Wiley, 2013. 117

[Bre09] Clay Breshears. *The art of concurrency: A thread monkey's guide to writing parallel applications.* O'Reilly Media, Inc., 2009. 33

[Bro96] Stephen D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996. 97

[Bro09] Paul Browne. *JBoss Drools business rules.* Packt Publishing, 2009. 55

[BS84] Bruce G. Buchanan and Edward H. Shortliffe. *Rule based expert systems: The MYCIN experiments of the Stanford heuristic programming project.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984. 55

[BSS12] Cristiano Bertolini, Martin Schäf, and Pascal Schweitzer. Infeasible code detection. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, pages 310–325, 2012. 149

[Bus12] Business Weekly. Software bugs cost more than double Eurozone bailout. http://www.businessweekly.co.uk/hi-tech/14898-software-bugs-cost-more-than-double-eurozone-bailout, December 2012. 21

[Cam08] Cambridge University Press. *Cambridge advanced learners dictionary.* Cambridge University Press, 3rd edition, 2008. 31

[CAS+10] Tudor Cioara, Ionut Anghel, Ioan Salomie, Mihaela Dinsoreanu, Georgiana Copil, and Daniel Moldovan. A reinforcement learning based self-healing algorithm for managing context adaptation. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '10, pages 859–862, New York, NY, USA, 2010. ACM. 37

[CASD09] Tudor Cioara, Ionut Anghel, Ioan Salomie, and Mihaela Dinsoreanu. A policy-based context aware self-management model. In *Proceedings of the 2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC '09, pages 333–340, Washington, DC, USA, 2009. IEEE Computer Society. 38

[CBC+04] Norman H. Cohen, James Black, Paul Castro, Maria Ebling, Barry Leiba, Archan Misra, and Wolfgang Segmuller. Building context-aware applications with context weaver. Technical Report RC23388, IBM Research, 2004. 33

[CC08] Ben Chelf and Andy Chou. Controlling software complexity. http://www.coverity.com/library/pdf/ControllingSoftwareComplexity.pdf, January 2008. 21

[CCK11] Hong-Zu Chou, Kai-Hui Chang, and Sy-Yen Kuo. Facilitating unreachable code diagnosis and debugging. In *Proceedings of the 16th Asia and*

South Pacific Design Automation Conference, ASPDAC '11, pages 485–490, Piscataway, NJ, USA, 2011. IEEE Press. 149

[CCP08] Javier Cubo, Carlos Canal, and Ernesto Pimentel. Towards a model-based approach for context-aware composition and adaptation: A case study using WF/.NET. In *Proceedings of the 2008 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, MOMPES '08, pages 3–13, Washington, DC, USA, 2008. IEEE Computer Society. 39

[CDKB11] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed systems: Concepts and design.* Addison-Wesley Publishing Company, USA, 5th edition, 2011. 173

[CdLG+09] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors. *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*. Springer, 2009. 36

[CFGO05] L. Cristaldi, M. Faifer, F. Grande, and R. Ottoboni. An improved M2M platform for multi-sensors agent application. In *Sensors for Industry Conference, 2005*, pages 79 –83, feb. 2005. 27

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking.* Mit Press, 1999. 128

[Cha03] S.K. Chang. *Data Structures and Algorithms.* Software Engineering and Knowledge Engineering, 13. World Scientific, 2003. 79

[Cha11] Dan Chalmers. *Sensing and systems in pervasive computing - Engineering context-aware systems.* Undergraduate Topics in Computer Science. Springer, London, UK, 2011. 32, 35

[Che10] M. Chemuturi. *Mastering software quality assurance: Best practices, tools and technique for software developers.* J Ross Publishing Series. J. Ross Pub., 2010. 22

[Chi10] Raymond Chiong, editor. *Intelligent systems for automated learning and adaptation: Emerging trends and applications.* IGI Global, 2010. 37

[Cho08] Jongmyung Choi. Software architecture for extensible context-aware systems. In *Proceedings of the 2008 International Conference on Convergence and Hybrid Information Technology*, ICHIT '08, pages 811–816, Washington, DC, USA, 2008. IEEE Computer Society. 36

[CK00] Guanling Chen and David Kotz. A survey of context-aware mobile computing research. Technical report, Dartmouth Computer Science Department, Hanover, NH, USA, 2000. 34

[CK08] Norman H. Cohen and Karl Trygve Kalleberg. EventScript: An event-processing language based on regular expressions with actions. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '08, pages 111–120, New York, NY, USA, 2008. ACM. 27, 48, 49, 180

[CLI13] CLIPS Expert System Group. CLIPS. `http://clipsrules.sourceforge.net/`, 2013. 56, 179

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 3rd edition, 2009. 76, 77

[CM98] Guy Cousineau and Michel Mauny. *The functional approach to programming.* Cambridge University Press, 1998. 120

[CM03] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 2003. 59

[CMKR11] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with Jolt. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 609–633, Berlin, Heidelberg, 2011. Springer-Verlag. 148

[CMP04] Carlos Canal, Juan Manuel Murillo, and Pascal Poizat. Coordination and adaptation techniques for software entities. In *ECOOP Workshops*, pages 133–147, 2004. 38

[CMP06] Carlos Canal, Juan Manuel, and Murillo Pascal Poizat. Software adaptation. In *L'objet, 12(1):9-31, 2006. Special Issue on Coordination and Adaptation Techniques for Software Entities*, pages 9–31, 2006. 38

[Coh07] Norman H. Cohen. EventScript: Using regular expressions to program event-processing agents. Technical Report RC 24387, IBM Research, October 2007. 48

[Coh08] Norman H. Cohen. Compound event processing using regular expressions: Examples from EventScript. Technical Report RC 24517, IBM Research, March 2008. 48

[Cor13] KUKA Robotics Corporation. Kuka. `http://www.kuka-robotics.com`, 2013. 162

[Cro11] James L. Crowley. Intelligent systems: Reasoning and recognition. `http://www-prima.imag.fr/Prima/Homepages/jlc/Courses/2010/ENSI2.SIRR/ENSI2.SIRR.S1.pdf`, February 2011. 55

[CSMP11] Francesco Chinello, Stefano Scheggi, Fabio Morbidi, and Domenico Prattichizzo. Kuka control toolbox. *IEEE Robot. Automat. Mag.*, 18(4):69–79, 2011. 162

[CT05] Richard H. Carver and Kuo-Chung Tai. *Modern multithreading: Implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs*. Wiley-Interscience, 2005. 33

[CT11] K. Cooper and L. Torczon. *Engineering a Compiler*. Elsevier Science, 2011. 123

[CTP+10] DeJiu Chen, Martin Törngren, Magnus Persson, Lei Feng, and Tahir Naseer Qureshi. Towards model-based engineering of self-

configuring embedded systems. In *Proceedings of the 2007 International Dagstuhl conference on Model-based engineering of embedded real-time systems*, MBEERTS'07, pages 345–353, Berlin, Heidelberg, 2010. Springer-Verlag. 36

[Cur11] K. Curran. *Ubiquitous developments in ambient computing and intelligence: Human-centered applications*. Igi Global, 2011. 32

[Dar09] Waltenegus Dargie. *Context-Aware Computing and Self-Managing Systems*. Chapman & Hall/CRC, 1 edition, 2009. 33, 34, 177

[DAS01] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comput. Interact.*, 16(2):97–166, December 2001. 31, 32

[Den12] Kerstin Denecke. *Event-driven surveillance: Possibilities and challenges*. Springer, Berlin, 2012. 153

[Dey00] Anind K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, Georgia Institute of Technology, Atlanta, November 2000. 32

[Dey01] Anind K. Dey. Understanding and using context. *Personal Ubiquitous Comput.*, 5(1):4–7, January 2001. 32

[DL05] Pierre-Charles David and Thomas Ledoux. WildCAT: A generic framework for context-aware applications. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, MPAC '05, pages 1–7, New York, NY, USA, 2005. ACM. 39

[DL10] G. Dowek and J.J. Lévy. *Introduction to the Theory of Programming Languages*. Undergraduate Topics in Computer Science. Springer, 2010. 118

[DSFv09] Laura M. Daniele, Eduardo Silva, Luis Ferreira, and Marten Sinderen van. A SOA-based platform-specific framework for context-aware mobile applications. In *Enterprise Interoperability*, volume 38 of *Lecture Notes in Business Information Processing*, pages 25–37, Berlin Heidelberg, 2009. Springer Verlag. 32

[DW08] Weichang Du and Lei Wang. Context-aware application programming for mobile devices. In *Proceedings of the 2008 C3S2E conference*, C3S2E '08, pages 215–227, New York, NY, USA, 2008. ACM. 39

[DZK+02] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, EW 10, pages 186–189, 2002. 39, 40, 180

[EJ09] Patrick Eugster and K. R. Jayaram. EventJava: An extension of Java for event correlation. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 570–594, Berlin, Heidelberg, 2009. Springer-Verlag. 40, 41, 43, 180

[EN10] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., 2010. 40

[Eng10] A. Engel. *Verification, Validation, and Testing of Engineered Systems*. Wiley Series in Systems Engineering and Management. Wiley, 2010. 128

[Eri11] Ericsson. More than 50 billion connected devices. `http://www.ericsson.com/res/docs/whitepapers/wp-50-billions.pdf`, 2011. 19, 25

[EV12] Benjamin J. Evans and Martijn Verburg. *The well-grounded Java developer: Vital techniques of Java 7 and polyglot programming*. Manning Publications Co., Greenwich, CT, USA, 2012. 64

[Fai06] Ted Faison. *Event-based programming: Taking events to the limit*. Apress, Berkely, CA, USA, 2006. 40

[Fai09] R.E. Fairley. *Managing and Leading Software Projects*. Wiley, 2009. 128

[FC09] Jorge Fox and Siobhán Clarke. Exploring approaches to dynamic adaptation. In *Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction*, MAI '09, pages 19–24, New York, NY, USA, 2009. ACM. 38

[FED12] FEDER. The MCUBE project. `http://www.systematic-paris-region.org/en/projets/mcube`, 2012. 24, 154

[Fed13] The Robocup Federation. Robocup's homepage. `http://www.robocup.org/`, 2013. 163

[Fer06] Stephen Ferg. Event-driven programming: Introduction, tutorial, history. `http://eventdrivenpgm.sourceforge.net`, 2006. 40

[FH12] Ernest Friedman-Hill. Jess. `http://www.jessrules.com/`, 2012. 57, 179

[Fis06] Marcus S. Fisher. *Software verification and validation: An engineering and scientific approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 127

[FM11] Samuel H. Fuller and Lynette I. Millett. *The future of computing performance: Game over or next level?* The National Academies Press, 2011. 64

[FMM07] Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. Tasks: Language support for event-driven programming. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '07, pages 134–143, New York, NY, USA, 2007. ACM. 45

[FMSP11] José Miguel Faria, J. Martins, and Jorge Sousa Pinto. An approach to model checking Ada programs. In *INFORUM 2011*, 2011. 131, 152

[Fow10] Martin Fowler. *Domain Specific Languages.* Addison-Wesley Professional, 1st edition, 2010. 23

[Fri06] J. Friedl. *Mastering regular expressions.* Oreilly Series. O'Reilly Media, Incorporated, 2006. 94

[FS03] Thomas Fahringer and Bernhard Scholz. *Advanced symbolic analysis for compilers: New techniques and algorithms for symbolic program analysis and optimization.* Springer-Verlag, Berlin, Heidelberg, 2003. 174

[FT12] Zhong Fan and Siok Tan. M2M communications for E-health: Standards, enabling technologies, and research challenges. In *Medical Information and Communication Technology (ISMICT), 2012 6th International Symposium on*, pages 1 –4, march 2012. 26

[Gab06] H.A. Gabbar. *Modern formal methods and applications.* Springer, 2006. 22

[Gar05] Simson Garfinkel. History's worst software bugs. `http://www.wired.com/software/coolapps/news/2005/11/69355`, August 2005. 21

[Gar13] Willow Garage. *OpenCV*, 2013. 72

[GBE+09] K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, S. Hallsteinsen, G. Horn, M. U. Khan, A. Mamelli, G. A. Papadopoulos, N. Paspallis, R. Reichle, and E. Stav. A comprehensive solution for application-level adaptation. *Softw. Pract. Exper.*, 39(4):385–422, March 2009. 38

[Ger97] Rob Gerth. Concise Promela reference. `http://www.spinroot.com/spin/Man/Quick.html`, 1997. 149

[GGGT09] Adrian Giurca, Adrian Giurca, Dragan Gasevic, and Kuldar Taveter. *Handbook of research on emerging rule-based languages and technologies: Open solutions and approaches.* IGI Publishing, Hershey, PA, USA, 2009. 55

[GHB+08] David Gouaillier, Vincent Hugel, Pierre Blazevic, Chris Kilner, Jérôme Monceaux, Pascal Lafourcade, Brice Marnier, Julien Serre, and Bruno Maisonnier. The Nao humanoid: A combination of performance and affordability. *CoRR*, abs/0807.3223, 2008. 163

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. 53

[GHJX08] Alex Groce, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. Putting flight software through the paces with testing, model checking, and constraint-solving. In *Proceedings of the Fifth International Workshop on Constraints in Formal Verification*, 2008. 142

[Gho10] Debasish Ghosh. *DSLs in action.* Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010. 23, 64, 180

[Gia07] Joseph C. Giarratano. *CLIPS user's guide*, dec 2007. 57

[GL09] J. Goyvaerts and S. Levithan. *Regular expressions cookbook.* O'Reilly Media, 2009. 94

[GMP04] María-Del-Mar Gallardo, Pedro Merino, and Ernesto Pimentel. A generalized semantics of Promela for abstract model checking. *Formal Aspects of Computing*, 16(3):166–193, August 2004. 97, 138

[Gos11] Gostai. *The Urbi software development kit*, jul 2011. 50, 180

[Gos13] Gostai Technologies. Urbi. `http://www.urbiforge.org/`, 2013. 50, 51, 52

[GP06] B. Goetz and T. Peierls. *Java concurrency in practice.* Addison-Wesley, 2006. 156

[gPh13] gPhoto group. gphoto2. `http://gphoto.org/`, 2013. 155

[GR98] J.C. Giarratano and G. Riley. *Expert systems: Principles and programming.* Computer Science Series. PWS Publishing Company, 1998. 57

[GvRB+12] D. Grune, K. van Reeuwijk, H.E. Bal, C.J.H. Jacobs, and K. Langendoen. *Modern Compiler Design.* Springer New York, 2012. 23

[Har00] Robert Harper. Type systems for programming languages, 2000. 118

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, New York, NY, USA, 1st edition, 2009. 124

[Has04] Patrik Haslum. Patterns in reactive programs. In *Proceedings of the Fourth International Cognitive Robotics Workshop*, 2004. 55

[Hav99] Klaus Havelund. Java PathFinder, a translator from Java to Promela. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, page 152, London, UK, 1999. Springer-Verlag. 131, 152

[HB07] Gerard J. Holzmann and Dragan Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Trans. Softw. Eng.*, 33:659–674, October 2007. 142

[HB10] Annika Hinze and Alejandro P. Buchmann, editors. *Principles and Applications of Distributed Event-Based Systems.* IGI Global, 2010. 40, 173

[HBE11] O. Hersent, D. Boswarthick, and O. Elloumi. *The Internet of things: Key applications and protocols.* John Wiley & Sons, Incorporated, 2011. 28

[Hen90] Matthew Hennessy. *The semantics of programming languages: An elementary introduction using structural operational semantics.* John Wiley and Sons, New York, N.Y., 1990. 97

[HHC11] Mahmoud Hussein, Jun Han, and Alan Colman. An approach to model-based development of context-aware adaptive systems. In *Proceedings of*

the 2011 IEEE 35th Annual Computer Software and Applications Conference, COMPSAC '11, pages 205–214, Washington, DC, USA, 2011. IEEE Computer Society. 37

[HHCY12] Mahmoud Hussein, Jun Han, Alan Colman, and Jian Yu. An architecture-based approach to developing context-aware adaptive systems. In *Engineering of Computer Based Systems (ECBS), 2012 IEEE 19th International Conference and Workshops on*, pages 154 –163, april 2012. 36

[Hil03] Ernest Friedman Hill. *Jess in action: Java rule-based systems.* Manning Publications Co., Greenwich, CT, USA, 2003. 55, 57, 58, 179

[HNS+09] A. Herstad, E. Nersveen, H. Samset, A. Storsveen, S. Svaet, and K.E. Husa. Connected objects: Building a service platform for M2M. In *Intelligence in Next Generation Networks, 2009. ICIN 2009. 13th International Conference on*, pages 1 –4, oct. 2009. 27

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, May 1997. 19, 130, 131

[Hol03] Gerard J. Holzmann. *The SPIN model checker - Primer and reference manual.* Addison-Wesley Professional, 1st edition, 2003. 130, 131, 138, 139

[HP85] D. Harel and A. Pnueli. *On the development of reactive systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985. 32

[HPSA10] Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic contract layers. In *25th Symposium on Applied Computing*, New York, NY, USA, 2010. ACM DL. 39

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008. 33

[HTW12] M. Teresa Higuera-Toledano and Andy J. Wellings, editors. *Distributed, embedded and real-time Java systems.* Springer, 2012. 156

[Hua09] J. C. Huang. *Software Error Detection through Testing and Analysis.* Wiley Publishing, 2009. 117

[HZJE11] Adrian Holzer, Lukasz Ziarek, K.R. Jayaram, and Patrick Eugster. Putting events in context: Aspects for event-based distributed programming. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 241–252, New York, NY, USA, 2011. ACM. 27

[IEE91] IEEE. IEEE standard computer dictionary - A compilation of IEEE standard computer glossaries. *IEEE Std 610*, page 1, 1991. 127

[Inc13] Logic Design Inc. Robologix. `http://www.robologix.com/programming_robologix.php`, 2013. 162

[INR13a] INRIA. Tom. `http://tom.loria.fr/tomplanet/`, 2013. 60

[INR13b] INRIA. Tom language documentation. `http://tom.loria.fr/wiki/index.php5/Documentation_Tom-2.10`, 2013. 60

[ISE13] ISEP. `http://www.isep.fr`, 2013. 64

[JBS11] C. Jones, O. Bonsignour, and J. Subramanyam. *The Economics of Software Quality.* Addison-Wesley, 2011. 21, 22

[JE09] K. R. Jayaram and Patrick Eugster. Context-oriented programming with EventJava. In *International Workshop on Context-Oriented Programming*, COP '09, pages 9:1–9:6, New York, NY, USA, 2009. ACM. 40, 44, 162

[JFM06] Rupak Majumdar Jeffrey Fischer and Todd Millstein. Preventing lost messages in event-driven programming. Technical Report TR060001, UCLA CSD, January 2006. 45, 46

[Jon10] Capers Jones. *Software Engineering Best Practices.* McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010. 23

[Kah87] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, UK, 1987. Springer-Verlag. 96

[KAK+10] Bo Hyun Kim, Hyeong-Joon Ahn, Jin Oh Kim, Myungsik Yoo, KyuJung Cho, and DongSoo Choi. Application of M2M technology to manufacturing systems. In *Information and Communication Technology Convergence (ICTC), 2010 International Conference on*, pages 519 –520, nov. 2010. 26

[Kat99] J.P. Katoen. *Concepts, Algorithms, and Tools for Model Checking.* Arbeitsberichte des Instituts für Mathematische Maschinen und Datenverarbeitung. Inst. für Mathematische Maschinen und Datenverarbeitung, 1999. 19, 128, 129

[KCKK08] Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. Formal verification of a flash memory device driver — An experience report. In *Proceedings of the 15th international workshop on Model Checking Software*, SPIN '08, pages 144–159, Berlin, Heidelberg, 2008. Springer-Verlag. 142

[KD06] Mohan Kumar and Sajal K Das. Pervasive computing: Enabling technologies and challenges. In AlbertY. Zomaya, editor, *Handbook of Nature-Inspired and Innovative Computing*, pages 613–631. Springer US, 2006. 32

[Ke09] Jiang Ke. Model checking C programs by translating C to Promela, 2009. 131, 152

[KGCC11] S.C. Kang, K.Y. Gu, W.T. Chang, and H.L. Chi. *Robot development using Microsoft Robotics Developer Studio.* Taylor & Francis, 2011. 162

[KK04] Claude Kirchner and Hélène Kirchner. Rule-based programming and proving: The ELAN experience outcomes. In *Proceedings of the 9th Asian Computing Science conference on Advances in Computer Science: dedicated to Jean-Louis Lassez on the Occasion of His 5th Cycle Birthday*, ASIAN'04, pages 363–379, Berlin, Heidelberg, 2004. Springer-Verlag. 55

[KM08] Fred Kröger and Stephan Merz. *Temporal logic and state systems (Texts in theoretical computer science).* Springer Publishing Company, Incorporated, 1 edition, 2008. 141

[Kru09] John Krumm. *Ubiquitous Computing Fundamentals.* Chapman & Hall/CRC, 1st edition, 2009. 32, 34, 39, 55

[KSK09] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data flow analysis: Theory and practice.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009. 174

[KT10] Devdatta Kulkarni and Anand Tripathi. A framework for programming robust context-aware applications. *IEEE Transactions on Software Engineering*, 36:184–197, 2010. 39

[LB07] Linda M. Laird and M. Carol Brennan. *Software measurement and estimation: A practical approach.* IEEE Computer Society, Washington, DC, USA, 2007. 21

[LBM10] Ivan Lanese, Antonio Bucchiarone, and Fabrizio Montesi. A framework for rule-based dynamic adaptation. In *Proceedings of the 5th international conference on Trustworthly global computing*, TGC'10, pages 284–300, Berlin, Heidelberg, 2010. Springer-Verlag. 55

[Le12] Truong-Giang Le. A demonstration video for programming robots with INI. `http://www.youtube.com/watch?v=alKZ9gZa4AU`, 2012. 169

[Lev12] A. Levitin. *Introduction to the Design and Analysis of Algorithms.* Pearson Education, 3rd edition, 2012. 76, 77

[LF05] Antónia Lopes and José Luiz Fiadeiro. Context-awareness in software architectures. In *Proceedings of the 2nd European conference on Software Architecture*, EWSA'05, pages 146–161, Berlin, Heidelberg, 2005. Springer-Verlag. 36

[LFH+13] Truong-Giang Le, Dmitriy Fedosov, Olivier Hermant, Matthieu Manceny, Renaud Pawlak, and Renaud Rioboo. Programming Robots with Events. In *Proceedings of the 4th International Embedded Systems Symposium (IESS 2013), Paderborn, Germany, 17 - 19 June, 2013*, pages 14–25, 2013. 164

[LHM+12] Truong-Giang Le, Olivier Hermant, Matthieu Manceny, Renaud Pawlak, and Renaud Rioboo. Unifying event-based and rule-based styles to develop concurrent and context-aware reactive applications - Toward a convenient support for concurrent and reactive programming. In *Proceedings of the 7th International Conference on Software Paradigm Trends, Rome, Italy, 24 - 27 July, 2012*, pages 347–350, 2012. 82, 83

[LHM+13] Truong-Giang Le, Olivier Hermant, Matthieu Manceny, Renaud Pawlak, and Renaud Rioboo. Using Event-based Style for Developing M2M Applications. In *Proceedings of the 8th International Conference on Grid and Pervasive Computing (GPC 2013), Seoul, Korea, 09 - 11 May, 2013*, pages 348–357, 2013. 154

[LHMP11] Truong-Giang Le, Olivier Hermant, Matthieu Manceny, and Renaud Pawlak. Dynamic adaptation through event reconfiguration. In Robert Meersaman, Tharam Dillon, and Pilar Herrero, editors, *On the Move to meaningful Internet Systems: OTM 2011 Workshops*, volume 7046 of *Lecture Notes in Computer Science*. Springer, 10 2011. 113

[LK11] Hyun Jung La and Soo Dong Kim. Static and dynamic adaptations for service-based systems. *Inf. Softw. Technol.*, 53(12):1275–1296, December 2011. 38

[LL11] K.C. Louden and K.A. Lambert. *Programming languages: Principles and practices.* Advanced Topics Series. Course Technology Ptr, 2011. 123, 171

[Llo84] John W. Lloyd. *Foundations of logic programming.* Springer, 1st edition, 1984. 59

[Lok06] Seng Loke. *Context-Aware Pervasive Systems.* Auerbach Publications, Boston, MA, USA, 2006. 32, 33, 34, 55

[LP12] Truong-Giang Le and Renaud Pawlak. *INI Project Online*, 2012. 64, 75, 93, 119, 141, 180, 190

[LS09] Janusz Laski and William Stanley. *Software verification and analysis: An integrated, hands-on approach.* Springer Publishing Company, Incorporated, 1 edition, 2009. 117, 128, 148

[LVS09] R. Lopes, D. Vicente, and N. Silva. Static analysis tools, a practical approach for safety-critical software verification. In *ESA Special Publication*, volume 669 of *ESA Special Publication*, May 2009. 117

[Mas03] Damien Massé. Property checking driven abstract interpretation-based static analysis. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI 2003, pages 56–69, London, UK, UK, 2003. Springer-Verlag. 174

[MB09] Basel Magableh and Stephen Barrett. PCOMs: A component model for building context-dependent applications. In *Proceedings of the 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, COMPUTATIONWORLD '09, pages 44–48, Washington, DC, USA, 2009. IEEE Computer Society. 37

[McC04] Steve McConnell. *Code Complete.* Microsoft Press, Redmond, WA, USA, 2nd edition, 2004. 23

[MCCDL06] Iqbal Mohomed, Jim Chengming Cai, Sina Chavoshi, and Eyal De Lara. Context-aware interactive content adaptation. In *Proceedings of the 4th*

international conference on Mobile systems, applications and services, MobiSys '06, pages 42–55, New York, NY, USA, 2006. ACM. 37

[MdRM09] Panos Markopoulos, Boris E. R. de Ruyter, and Wendy E. Mackay, editors. *Awareness systems - Advances in theory, methodology and design.* Human-Computer Interaction Series. Springer, 2009. 33

[Med08] A. Meduna. *Elements of Compiler Design.* Computer science. Computer engineering. Computing. Auerbach Publications, 2008. 127

[Mer01] Stephan Merz. Model checking: A tutorial overview. In *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, MOVEP '00, pages 3–38, London, UK, UK, 2001. Springer-Verlag. 128

[MFP06] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 39, 40, 173, 180

[MG08] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. *SIGPLAN Not.*, 43(1):51–62, January 2008. 97

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005. 23

[Mit03] J.C. Mitchell. *Concepts in Programming Languages.* Cambridge University Press, 2003. 123

[MM11] E.T. Matson and Byung-Cheol Min. M2M infrastructure to integrate humans, agents and robots into collectives. In *Instrumentation and Measurement Technology Conference (I2MTC), 2011 IEEE*, pages 1 –6, may 2011. 27

[MM12] Markets and Markets. Service robotics market (Personal & professional) - Global forecast & assessment by applications & geography 2012-2017, 2012. 161

[Mob09] S. Saeidi Mobarakeh. Type inference algorithms. `http://www.win.tue.nl/~hzantema/semssm.pdf`, 2009. 124

[Moh08] C. Mohan. *Design And Analysis Of Algorithms*. Prentice-Hall Of India Pvt. Limited, 2008. 79

[MTFA11] Moeiz Miraoui, Chakib Tadj, Jaouhar Fattahi, and Chokri Ben Amar. Dynamic context-aware and limited resources-aware service adaptation for pervasive computing. *Adv. Soft. Eng.*, 2011:7:7–7:7, January 2011. 35

[Mun09] V.V. Muniswamy. *Design And Analysis Of Algorithms*. I.K. International Publishing House Pvt. Ltd., 2009. 79

[NAS12] NASA. OPS-2000. `http://www.siliconvalleyone.com/founder/ops2000/index.htm`, 2012. 59, 60

[Nat12] Suman Nath. ACE: Exploiting correlation for energy-efficient and continuous context sensing. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 29–42, New York, NY, USA, 2012. ACM. 39

[NDR09] Oscar Nierstrasz, Marcus Denker, and Lukas Renggli. Model-centric, context-aware software adaptation. In Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, chapter Model-Centric, Context-Aware Software Adaptation, pages 128–145. Springer-Verlag, Berlin, Heidelberg, 2009. 37

[Net11a] Network Computing. Cisco jumps into the M2M market. `http://www.networkcomputing.com/wireless/231600077`, 2011. 25

[Net11b] Juniper Networks. Machine-to-machine (M2M) - The rise of the machines. `http://www.juniper.net/us/en/local/pdf/whitepapers/2000416-en.pdf`, 2011. 27

[NH96a] V. Natarajan and Gerard J. Holzmann. Outline for an operational semantics of PROMELA. In *The SPIN Verification System. Proceedings of the Second SPIN Workshop 1996., volume 32 of DIMACS. AMS*, 1996. 97

[NH96b] V. Natarajan and Gerard J. Holzmann. Outline for an operational semantics of Promela. In *The SPIN Verification System. Proceedings of the Second SPIN Workshop 1996., volume 32 of DIMACS. AMS*, 1996. 138

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: A formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992. 97

[NSHW10] Evi Nemeth, Garth Snyder, Trent R. Hein, and Bent Whaley. *UNIX and Linux System Administration Handbook.* Prentice Hall, 4th edition, 2010. 69

[NYS+05] Kouji Nishigaki, Keiichi Yasumoto, Naoki Shibata, Minoru Ito, and Teruo Higashino. Framework and rule-based language for facilitating context-aware computing using information appliances. In *Proceedings of the First International Workshop on Services and Infrastructure for the Ubiquitous and Mobile Internet (SIUMI) (ICDCSW'05) - Volume 03*, ICDCSW '05, pages 345–351, Washington, DC, USA, 2005. IEEE Computer Society. 55

[Obe05] R. Obermaisser. *Event-triggered and time-triggered control paradigms.* Real-Time Systems. Springer, 2005. 40

[Ora13a] Oracle. *Java SE 6 documentation*, 2013. 114

[Ora13b] Oracle. Oracle Java SE embedded. `http://www.oracle.com/technetwork/java/embedded/overview/getstarted/index.html`, 2013. 156

[Ora13c] Oracle. Regular Expressions in Java. `http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html`, 2013. 94

[Ort12] S. Ortiz. Computing trends lead to new programming languages. *Computer*, 45(7):17 –20, july 2012. 23

[OxSD+09] Jian-Quan Ouyang, Dian xi Shi, Bo Ding, Jin Feng, and Huaimin Wang. Policy based self-adaptive scheme in pervasive computing. *Wireless Sensor Network*, 1(1):48–55, 2009. 38

[Par09] Terence Parr. *Language implementation patterns: Create your own domain-specific and general programming languages.* Pragmatic Bookshelf, 1st edition, 2009. 23

[Pat07] Srikanta Patnaik. *Robot cognition and navigation - An experiment with mobile robots.* Cognitive Technologies. Springer, 2007. 164

[PBHS03] A. Pashtan, R. Blattler, A. Heusser, and P. Scheuermann. CATIS: A context-aware tourist information system. In *Proceedings of the 4th International Workshop of Mobile Computing (IMC'03)*, Rostock, Germany, 2003. 39

[Pie02] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002. 118

[Pie04] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages.* The MIT Press, 2004. 118

[Plo81] G. D. Plotkin. A structural approach to operational semantics, 1981. 97

[Plo04] Gordon D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004. 97

[Pol10] Polytechnic University of Catalonia. CLIPS - Code snippets. `http://www.lsi.upc.edu/~bejar/ia/material/laboratorio/clips/CLIPS-snippets-eng.pdf`, 2010. 56

[Pre05] Scott Preston. *The definitive guide to building Java robots.* Apress, Berkely, CA, USA, 2005. 162

[Pre10] R.S. Pressman. *Software engineering: A practitioner's approach.* McGraw-Hill higher education. McGraw-Hill Higher Education, 2010. 174

[Rak03] Andry Rakotonirainy. How to program pervasive systems. In *DEXA Workshops*, pages 947–948, 2003. 39

[RC02] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 205–230, London, UK, UK, 2002. Springer-Verlag. 38

[Res11a] Beecham Research. M2M sector map. `http://www.beechamresearch.com/`, 2011. 26

[Res11b] Juniper Research. M2M to generate $35bn in service revenues by 2016. `http://juniperresearch.com/viewpressrelease.php?pr=243`, 2011. 25

[RH04] Theo C. Ruys and Gerard J. Holzmann. Advanced SPIN tutorial. In *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings*, pages 304–305, 2004. 142

[RJSF06] Danny Raz, Arto Juhola, and Joan Serrat-Fernandez. *Fast and efficient context-aware services.* Wiley series in communications networking & distributed systems. Wiley, Hoboken, NJ, 2006. 34

[RN09] Stuart Russell and Peter Norvig. *Artificial intelligence: A modern approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. 161

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965. 124, 190

[Rob13] Robomatter. RobotC. `http://www.robotc.net/`, 2013. 162

[Roe11] K. Roebuck. *Machine-to-machine (M2M) communication services: High-impact technology - What you need to know: Definitions, adoptions, impact, benefits, maturity, vendors.* Lightning Source Incorporated, 2011. 28

[Roz11] Kristin Y. Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163 – 203, 2011. 141

[RPS05] Anca Rarau, Kalman Pusztai, and Ioan Salomie. Software framework for building context-aware applications using multifacet items. In *Proceedings of the 2nd International Workshop on Software Aspects of Context (IWSAC5)*, 2005. 55

[RS06] Anca Rarau and Ioan Salomie. Adding context awareness to C#. In *Proceedings of the First European conference on Smart Sensing and Context*, EuroSSC'06, pages 98–112, Berlin, Heidelberg, 2006. Springer-Verlag. 39, 177

[RS08] Daniel Retkowitz and Mark Stegelmann. Dynamic adaptability for smart environments. In *Proceedings of the 8th IFIP WG 6.1 international conference on Distributed applications and interoperable systems*, DAIS'08, pages 154–167, Berlin, Heidelberg, 2008. Springer-Verlag. 38

[RV01] J.J.A. Robinson and A. Voronkov. *Handbook of Automated Reasoning.* Number vol. 1 in Handbook of Automated Reasoning. Elsevier, 2001. 124

[Sah06] Goutam Kumar Saha. Software based fault tolerance: A survey. *Ubiquity*, 2006(July):1, July 2006. 173

[San11] B. Sandén. *Design of multithreaded software: The entity-life modeling approach.* John Wiley & Sons, 2011. 33

[Sat09] Ichiro Satoh. A context-aware service framework for large-scale ambient computing environments. In *Proceedings of the 2009 international conference on Pervasive services*, ICPS '09, pages 199–208, New York, NY, USA, 2009. ACM. 32

[SAW94] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, WMCSA '94, pages 85–90, Washington, DC, USA, 1994. IEEE Computer Society. 32, 177

[SBS+11] Simone Souza, Maria Brito, Rodolfo Silva, Paulo Souza, and Ed Zaluska. Research in concurrent software testing: A systematic review. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD 2011)*, pages 1–5. Conference Publishing Solutions, July 2011. invited paper. 33

[Sch08] Gregor Schmidt. ContextR & ContextWiki. Master's thesis, Hasso-Plattner-Institut, Potsdam, April 2008. 39

[Sco09] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2009. 118

[Seb12] R.W. Sebesta. *Concepts of Programming Languages*. Always learning. Pearson College Division, 2012. 127

[SGP12] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *J. Syst. Softw.*, 85(8):1801–1817, August 2012. 33, 39

[SH11] S. Singh and Kuei-Li Huang. A robust M2M gateway for effective integration of capillary and 3GPP networks. In *Advanced Networks and Telecommunication Systems (ANTS), 2011 IEEE 5th International Conference on*, pages 1 –3, dec. 2011. 153

[SHX11] Alexandra Poulovassilis Sven Helmer and Fatos Xhafa, editors. *Reasoning in Event-Based Distributed Systems*. Springer Berlin / Heidelberg, 2011. 40

[SK95] Kenneth Slonneger and Barry Kurtz. *Formal syntax and semantics of programming languages: A laboratory based approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995. 97

[Smi06] Joshua B. Smith. *Practical OCaml (Practical)*. Apress, Berkely, CA, USA, 2006. 91

[Sos09] Barrie Sosinsky. *Networking bible*. Wiley Publishing, 1st edition, 2009. 153

[Spi08] Michael Spivey. *An introduction to logic programming through Prolog*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2008. 59

[SPI13] Spin. `http://www.spinroot.com/`, 2013. 130

[SS06] C. Soanes and A. Stevenson. *Concise Oxford English dictionary*. Oxford University Press, 2006. 31

[SS13] Stefan Staiger-Stöhr. Practical integrated analysis of pointers, dataflow and control flow. *ACM Trans. Program. Lang. Syst.*, 35(1):5:1–5:48, April 2013. 174

[ST04] Alan Shalloway and James Trott. *Design patterns explained: A new perspective on object-oriented design*. Addison-Wesley Professional, 2nd edition, 2004. 53

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009. 33, 35, 38

[Sto89] J. Stobo. *Problem Solving With Prolog*. Taylor & Francis, 1989. 59

[SWH12] H. Seidl, R. Wilhelm, and S. Hack. *Compiler design: Analysis and transformation*. SpringerLink : Bücher. Springer, 2012. 117

[TC07] Shiu Lun Tsang and Siobhan Clarke. Mining user models for effective adaptation of context-aware applications. In *Proceedings of the The 2007 International Conference on Intelligent Pervasive Computing*, IPC '07, pages 178–187, Washington, DC, USA, 2007. IEEE Computer Society. 37

[Tei12] P. Teixeira. *Professional Node.js: Building Javascript based scalable software*. Wiley, 2012. 39

[The13] The Eclipse Foundation. Eclipse IDE. http://www.eclipse.org/, 2013. 173

[Tho96] Simon Thompson. *Haskell - The craft of functional programming*. International computer science series. Addison-Wesley, 1996. 120

[Tia05] J. Tian. *Software quality engineering: Testing, quality assurance, and quantifiable improvement*. John Wiley & Sons, 2005. 22

[Tuu00] Esa Tuulari. Context aware hand-held devices. Technical report, Technical Research Centre of Finland, 2000. 32

[VAELMnG+08] Javier Vales-Alonso, Esteban Egea-López, Juan Pedro Muñoz Gea, Joan García-Haro, Felix Belzunce-Arcos, Marco Antonio Esparza-García, Juan Manuel Pérez-Mañogil, Rafael Martínez-Álvarez, Felipe Gil-Castiñeira, and Francisco J. González-Castaño. UCare: Context-aware services for disabled users in urban environments. In *Proceedings of the 2008 The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, UBICOMM '08, pages 197–205, Washington, DC, USA, 2008. IEEE Computer Society. 36

[Wad07] V.K. Wadhawan. Robots of the future. *Resonance*, 12:61–78, 2007. 161

[Wal13] Timothy Wall. Java Native Access. `https://github.com/twall/jna#readme`, 2013. 73

[Wat05] A. Watt. *Beginning Regular Expressions.* Wiley India Pvt. Limited, 2005. 94

[Web13] Webdyn. Webdyn's homepage. `http://www.webdyn.com/en/`, 2013. 156

[Wei97a] Carsten Weise. An incremental formal semantics for PROMELA. In *In Proceedings of the Third SPIN Workshop, SPIN97*, 1997. 97

[Wei97b] Carsten Weise. An incremental formal semantics for Promela. In *In Proceedings of the Third SPIN Workshop, SPIN97*, 1997. 138

[Win93] Glynn Winskel. *The formal semantics of programming languages: An introduction.* MIT Press, Cambridge, MA, USA, 1993. 96, 97

[WMSC11] Hongyuan Wang, Rutvij Mehta, Sam Supakkul, and Lawrence Chung. Rule-based context-aware adaptation using a goal-oriented ontology. In *Proceedings of the 2011 International workshop on Situation activity and goal awareness*, SAGAware '11, pages 67–76, New York, NY, USA, 2011. ACM. 55

[WT10] F. M. Wahl and U. Thomas. Robot programming - From simple moves to complex robot tasks. `http://www2.cs.siu.edu/~hexmoor/classes/CS404-S10/Wahl.pdf`, 2010. 19, 162

[YW03] Hongji Yang and Martin Ward. *Successful Evolution of Software Systems.* Artech House, Inc., Norwood, MA, USA, 2003. 22

[Zen08] Jia Zeng. *Partial evaluation for code generation from domain-specific languages.* PhD thesis, New York, NY, USA, 2008. AAI3305282. 174

[ZKSL09] Yanmin Zhu, Sye Loong Keoh, M. Sloman, and E. C. Lupu. A lightweight policy system for body sensor networks. *IEEE Trans. on Netw. and Serv. Manag.*, 6(3):137–148, September 2009. 38

Appendixes

Acronyms

API Application Programming Interface. 65, 72, 95, 157, 163

AST Abstract Syntax Tree. 119, 127, 142

CAGR Compound Annual Growth Rate. 161

CCTV Closed-circuit Television. 27

CNC Computer(ized) Numerical(ly) Control(led). 161

CPS Continuation Passing Style. 45

CSV Comma-separated Values. 71, 158

DSL Domain Specific Language. 23, 28, 162

FTP File Transfer Protocol. 155, 156, 158

GPL General Purpose Language. 23

HTTP Hypertext Transfer Protocol. 45

HVAC Heating, Ventilation, and Air Conditioning. 26

I/O Input/Output. 45, 46, 53, 65, 152

IDE Integrated Development Environment. 173

JVM Java Virtual Machine. 64

LHS left-hand side. 56–60

LOC lines of code. 21

LTL Linear Temporal Logic. 130, 138, 139, 141, 149, 150, 159

M2M Machine-to-Machine. 24, 25, 27–30, 153, 157, 158

RHS right-hand side. 56–60

SoC Separation of Concerns. 38

V&V Verification and Validation. 127, 128

Index