Programming Robots With Events

Truong Giang Le, Dmitriy Fedosov, Olivier Hermant, Matthieu Manceny, Renaud Pawlak, Renaud Rioboo

Programming Robots with Events

Truong-Giang Le[1], Dmitriy Fedosov[2], Olivier Hermant[3], Matthieu Manceny[1], Renaud Pawlak[4], and Renaud Rioboo[5]

[1] LISITE - ISEP, 28 rue Notre-Dame des Champs, 75006 Paris, France
[2] Saint-Petersbourg University of Aerospace Instrumentation, 67 Bolshaya Morskaya street, 190000, Saint Petersburg, Russia
[3] CRI - MINES ParisTech, 35 rue ST-Honoré, 77300 Fontainebleau, France
[4] IDCapture, 2 rue Duphot, 75001 Paris, France
[5] ENSIIE, 1 square de la Résistance, F-91025 Évry CEDEX, France
{le-truong.giang,matthieu.manceny}@isep.fr,
{dvfdsv,renaud.pawlak}@gmail.com,
olivier.hermant@mines-paristech.fr,
renaud.rioboo@ensiie.fr

Abstract. We introduce how to use event-based style to program robots through the INI programming language. INI features both built-in and user-defined events, a mechanism to handle various kinds of changes happening in the environment. Event handlers run in parallel either synchronously or asynchronously, and events can be reconfigured at runtime to modify their behavior when needed. We apply INI to the humanoid robot called Nao, for which we develop an object tracking program.

Keywords: robotics, event-based programming, context-aware reactive systems, parallel programming.

1 Introduction

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, 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 the robots, since they are considered as a subset of "smart structures" - engineered constructs equipped with sensors to "think" and to adapt to the environment [28]. Generally, robots can be put into three main categories: manipulators, mobile robots and humanoid robots [26].

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 CNC (Computer(ized)

G. Schirner et al. (Eds.): IESS 2013, IFIP AICT 403, pp. 14–25, 2013.

Numerical(ly) Control(led)) 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 maybe also good friends for old people at home for talking and sharing housework. The global service robotics market in 2011 was worth $18.39 billion. This market is valued at $20.73 billion in 2012 and expected to reach $46.18 billion by 2017 at an estimated CAGR (Compound Annual Growth Rate) of 17.4% from 2012 to 2017 [20]. As a result, research on robot gets an increasing interests from governments, companies, and researchers [2,3].

Building a robot program 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.

We give an overview of such robot programming languages and frameworks in Section 2. Next, in Section 3, we discuss how to define and apply events in our novel programming language called INI, especially its advanced features like events synchronization and reconfiguration. Then, we present a case study of using INI to control the humanoid robot Nao to track an object (Section 4). Finally, some conclusions and future work are discussed in Section 5.

2 Related Work

Classical languages like Java, C/C++, .Net are usually used for programming robot [1,17,22]. However, developing robot's applications using these languages require more time and effort since they are not fully dedicated to this purpose. For example, to support interaction between robots and 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 may use some extensions like [16,23] although they are not easy to adapt.

Additionally, several robotic development platforms and DSLs (Domain Specific Languages) have been designed to assist programmers. Urbiscript is a scripting language primarily designed for robotics. It's a dynamic, prototype-based, and object-oriented scripting language. It supports and emphasizes parallel and event-based programming, which are very popular paradigms in robotics, by providing core primitives and language constructs [12]. UrbiScript has some limitations. First, it is an untyped language. Besides, it lacks support for synchronization among events, which is essential in some scenarios. Moreover, events in UrbiScript cannot be reconfigured at runtime to change their behavior.

The KUKA Robot Programming Language is developed by KUKA, one of the world's leading manufacturers of industrial robots [6]. KUKA is simple, Pascal-like and lacks many features. A Matlab abstraction layer has been introduced to extend its capabilities [4]. RoboLogix 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 [15]. The

data objects reside in registers and the program flow represents the list of instructions, or instruction set, that is used to program the robot. However, RoboLogix still does not supply a well-defined mechanism for the robots to interact and react to environment.

In recent years, event-driven programming has emerged as an efficient method for interacting and collaborating with the environment in ubiquitous computing. Using event-driven style requires less effort and may lead to more robust software [8]. This style is used to write many kinds of applications: robotics, context-aware reactive applications, self-adaptive systems, interactive systems, etc. In consequence, several event-based programming languages have been developed so far [5,14]. However, in these languages, events still cannot be defined intuitively and straightforwardly, and their features are still limited. For example, events cannot run in parallel to take advantage of multiple processors. Besides, programmers may not dynamically customize events' behavior to deal with changes in the environment.

Considering limitations of current event-based programming language, we have developed a novel programming language called INI. With INI, developers may define and use events easily. Along with several built-in events, they also can write their own events in Java or in C/C++, and then integrate to INI programs. Events in INI may run concurrently either asynchronously or synchronously. Moreover, events may be reconfigured at run time to handle different scenarios happening in the context.

3 Event-Based Programming with INI

3.1 Overview

Events are 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. Generally, there are three types of events [21]:

- A timer event to express the passing of time.
- An arbitrary detectable state change in the system, e.g. the change of the value of a variable during execution.
- A physical event such as the appearance of a person detected by cameras.

For example, programmers may define an event to monitor the power level of their systems or to observe users' behavior in order to react. They can also specify an event to schedule a desired action at preferable time. To understand more about event-based programming, please refer to [9,10].

INI is a programming language developed by ourselves, which runs on Java Virtual Machine (JVM) but INI's syntax and semantics are not Java's ones. INI is developed aiming at supporting the development of concurrent and context-aware reactive systems, which need a well-defined mechanism for capturing and handling events. As shown later, INI supports all those kinds of event. Event callback handlers (or events instances) are declared in the body of functions

and are raised, by default asynchronously, every time the event occurs. By convention, an event instance in INI starts with @ and 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 then the event callback is called, and executed in a new thread. They can be considered as the measured characteristic of the event instance. It has to be noticed that those variables, as well as any INI variable, enjoy a global scope in the function's body. Both kinds of parameters are optional. Moreover, an event can also be optionally bound to an id, so that other parts of the program can refer to it. The syntax of event instances is shown below:

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

Table 1. Some built-in events in INI

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 UNIX CRON pattern indicated by its input parameter.

Programmers may use built-in events (listed in Table 1), or write user-defined events (in Java or in C/C++), and then integrate them to their INI programs through bindings. By developing custom events, one can process data which are captured by sensors. To illustrate events in INI, let's 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 Figure 1. For instance, we can define an event `@humidityMonitor` 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. Other events can be defined in a similar structure. The last event is a built-in `@cron` event, which is employed to send these data to a server at 3:00 AM, 11:00 AM, and 7:00 PM every day (to learn more about UNIX CRON pattern, please refer to [7]). All events in our program run in parallel so that it can handle multiple tasks at one time.

```
function main() {
 h:@humidityMonitor[humPeriod = 1](humidity) {
  case {
   humidity > ... {...}
   default {...}
  }
 }
 t:@temperatureMonitor[tempPeriod = 2](temperature) { ... }
 ...
 @cron[pattern = "0␣3-11-19␣*␣*␣*"]() {
  //Send data to a server for tracking purpose ...
 }
}
```

Fig. 1. A sample INI program used for collecting climate data

3.2 Advanced Use of Events

By default, except for the `@init` and `@end` events (see Table 1), 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 running threads corresponding to the target events to be terminated before running. For instance, when `e0` may affect the actions defined inside other events, we need to apply the synchronization mechanism. The syntax corresponding to the above discussion is:

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

Events in INI may be reconfigured at runtime in order to adjust their behavior when necessary to adapt to changes happening in the environment. Programmers may use the built-in function `reconfigure_event(eventId, [inputParam1= value1, inputParam2=value2,...])` in order to modify the values of input parameters of the event referred to by `eventId`. For instance, in the example of Figure 1, we can call `reconfigure_event(h, [humPeriod=0.5])` to set the humidity data collection period to 30 minutes. Now our event will gather data every 30 minutes instead of one hour as before. Besides, we also allow programmers to stop and restart events with two built-in functions: `stop_event([eventId1, eventId2,...])` and `restart_event([eventId1, eventId2,...])`. For example, we can stop all data collection processes when the energy level of the system is too low, and restart them later when the energy is restored.

Last but not least, events in INI may be used in combination with a boolean expression to express the requirement that need to be satisfied before they can be executed. Programmers may use the syntax below:

```
<event_expression> <logical_expression> { <action> }
```

For example, if we want the event `@humidityMonitor` to be executed only when the temperature is higher than some threshold:

```
@humidityMonitor[humPeriod=1](humidity) temperature>... {...}
```

To understand more about the above mechanisms and other aspects of INI (e.g. developing user-defined events, rules, type system, type checking, and built-in functions), the readers may have a look at [19,27].

4 A Case Study with the Humanoid Robot Nao

In this section, we briefly present the humanoid robot Nao, especially its features related to the moving mechanism. Then we show an INI tracking program running on Nao. The purpose of our INI program is controlling the Nao to detect a ball in the space and then walk to reach it.

4.1 Introduction to Nao and Its Moving Mechanism

Nao is the humanoid robot that is built by the French company Aldebaran-Robotics [13,25]. It is equipped with many sensor devices to obtain robot's close environment information (see Figure 2). Nao has for instance become a standard platform for RoboCup, an international initiative that fosters research in robotics and artificial intelligence [11].

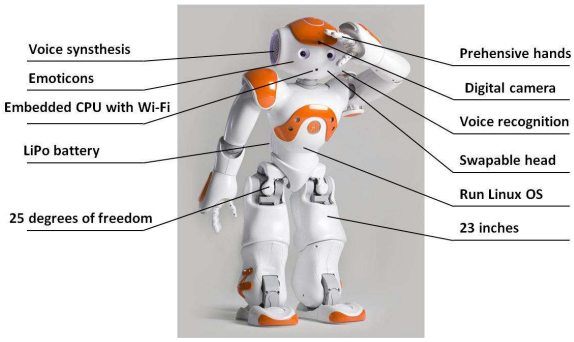

Fig. 2. Nao's features [25]

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 Windows, Mac or Linux, and be called from 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 several low-level accesses. They also provide a well-organized documentation, particularly on how to control the robot effectively [24].

Nao is able to walk on multiple floor surfaces such as carpet, tiles and wooden floors. Each step is composed of a double leg and a single leg support phase. With Nao, the basic foot step planner is used by the walk process [24], provided by three possible walk control APIs: `ALMotionProxy::setWalkTarget`

`Velocity()` (applied in our later case study), `ALMotionProxy :: walkTo()` or `ALMotionProxy::setFootSteps()`. The foot's position is specified by three parameters: `x`, `y` and θ (see Figure 3). `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]. The movement is composed of a translation by `x` and `y`, then a rotation around the vertical `Z` axis θ. It is possible to define custom gait parameters for the walk of Nao so that we can control the direction and speed to adjust to different scenarios. To learn more about these parameters (e.g. `MaxStepX`, `MaxStepY`, `MaxStepTheta`, `MaxStepFrequency`, etc.) along with their value ranges and default values, please refer to Nao's documentation [24]. In INI, we abstract over many of those parameters through user-defined events and functions in order to facilitate programming.

4.2 An INI Tracking Program Running on Nao

In this part, we show how INI can be applied for programming robot through the example of building a ball tracking program. Figure 3 displays the possible relative positions between the robot and the ball. There are three distinguished

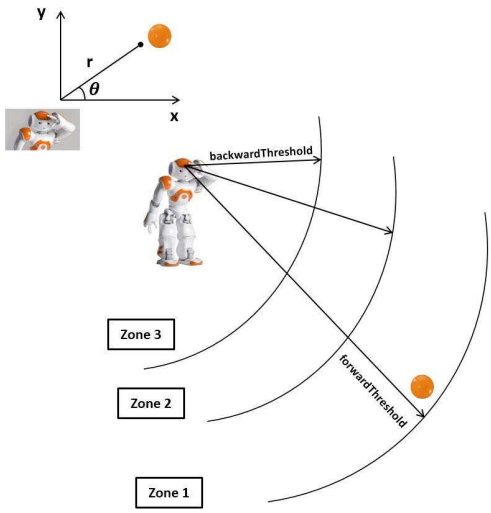

Fig. 3. Possible relative positions among the robot and the ball

zones that are specified based on the distance from the robot to the detected ball. And then according to which zone the ball belongs to, we can control the robot with the desired behavior:

– 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 `backwardThreshold` (its unit and range are the same as `forwardThreshold`) and `forwardThreshold`, the robot does not move since its 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 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 4. Our program is shown

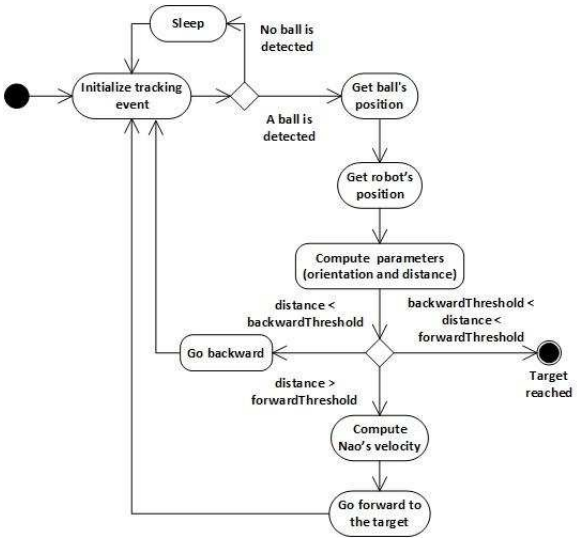

Fig. 4. The activity diagram for our program

in Figure 5. In our program, we employ three events. The event `@init` (lines 2-16) is applied to initialize the variables used later in our program. The purpose of using 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 [24]) 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 [24] 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

```
function main() {
 @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
 }
 $(e) d:@detectBall[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]
   stepX = parameters[2]
   i = 0
   needAdjustDirection = true
   stepFrequency = defaultStepFrequency
 }
 $(d,e) e:@every[time = 200]() {
  //Control the robot to go one step if the ball is detected
   needAdjustDirection = reach_to_target(name, port,
   stepFrequency, robotPosition, stepX, targetTheta,
    needAdjustDirection, useSensors)
  i++
  case {
   //Reset parameters after three consecutive walking steps
   i>3 {
    stepX = 0.0
    targetTheta = 0.0
    stepFrequency = 0.0
   }
  }
 }
}
```

Fig. 5. An object tracking program written in INI

that then we can calculate appropriate needed direction and speed for its movement. `stepX` is the fraction (between 0.0 and 1.0) of `MaxStepX` (the maximum translation along the `X` axis for one step, see [24]). The sign of `stepX` also indicates the moving direction (forward or backward) of the 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.

The event `@detectBall` (lines 17-28) is a user-defined event written in Java, 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 own. Inside this event, when a ball is detected, we call the 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 event `@every` (lines 29-43) is applied to control the robot to move towards the target every 200 milliseconds. The 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 temporary 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, 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 `@detectBall` 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 17 and line 29). Additionally, the event `@every` is also synchronized with itself so that each robot step does not overlap with others.

When running in experiment, our program completes well the desired requirements. The robot detects the orange ball in the space and then follows it. When the ball is moved to another position, Nao also changes the direction and speed to reach the ball if needed. A demonstration video can be watched on YouTube [18].

5 Conclusion and Future Work

In this paper, we presented how to write robot applications by using INI, a novel programming language that supports event-based paradigm. Programmers may use built-in events, or develop custom events in other languages like Java or C/C++ and then integrate them to INI programs. Moreover, events may run in parallel (asynchronously or synchronously) to speed up the execution and improve performance. Last but not least, in case of changes happening in the environment, events can be dynamically reconfigured to adapt to a new context.

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 the human voice commands, and then control the robot to act the desired behavior.

Acknowledgments. The work presented in this article is co-funded by the European Union. Europe is committed in Ile-de-France with the European Regional Development Fund.

References

1. Auyeung, T.: Robot programming in C (2006), http://www.drtak.org/teaches/ARC/cisp299_bot/book/book.pdf
2. Bar-Cohen, Y., Hanson, D.: The Coming Robot Revolution: Expectations and Fears About Emerging Intelligent, Humanlike Machines, 1st edn. Springer Publishing Company, Incorporated (2009)
3. Bekey, G.: Robotics: State of the Art and Future Challenges. World Scientific (2008)
4. Chinello, F., Scheggi, S., Morbidi, F., Prattichizzo, D.: Kuka control toolbox. IEEE Robot. Automat. Mag. 18(4), 69–79 (2011)
5. Cohen, N.H., Kalleberg, K.T.: 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 2008, pp. 111–120. ACM, New York (2008)
6. KUKA Robotics Corporation: Kuka, http://www.kuka-robotics.com
7. Crontab, http://crontab.org/
8. Dabek, F., Zeldovich, N., Kaashoek, F., Mazières, D., Morris, R.: Event-driven programming for robust software. In: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop, EW 10, pp. 186–189 (2002)
9. Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications Co. (2010)
10. Faison, T.: Event-Based Programming: Taking Events to the Limit. Apress, Berkely (2006)
11. Federation, T.R.: Robocup's homepage, http://www.robocup.org/
12. Gostai: The Urbi Software Development Kit (July 2011)

13. Gouaillier, D., Hugel, V., Blazevic, P., Kilner, C., Monceaux, J., Lafourcade, P., Marnier, B., Serre, J., Maisonnier, B.: The Nao humanoid: A combination of performance and affordability. CoRR abs/0807.3223 (2008)
14. Holzer, A., Ziarek, L., Jayaram, K., Eugster, P.: Putting events in context: aspects for event-based distributed programming. In: Proceedings of the Tenth International Conference on Aspect-Oriented Software Development, AOSD 2011, pp. 241–252. ACM, New York (2011)
15. Logic Design Inc.: Robologix, http://www.robologix.com/programming_robologix.php
16. Jayaram, K.R., Eugster, P.: Context-oriented programming with EventJava. In: International Workshop on Context-Oriented Programming, COP 2009, pp. 9:1–9:6. ACM, New York (2009)
17. Kang, S., Gu, K., Chang, W., Chi, H.: Robot Development Using Microsoft Robotics Developer Studio. Taylor & Francis (2011)
18. Le, T.G.: A demonstration video, http://www.youtube.com/watch?v=alKZ9gZa4AU
19. Le, T.G., Hermant, O., Manceny, M., Pawlak, R., Rioboo, R.: 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, July 24-27, pp. 347–350 (2012)
20. M&M: Service robotics market (personal & professional) global forecast & assessment by applications & geography (2012 - 2017) (2012), http://www.marketsandmarkets.com/
21. Mühl, G., Fiege, L., Pietzuch, P.: Distributed Event-Based Systems. Springer-Verlag New York, Inc., Secaucus (2006)
22. Preston, S.: The Definitive Guide to Building Java Robots (The Definitive Guide to). Apress, Berkely (2005)
23. Robomatter: Robotc, http://www.robotc.net/
24. Aldebaran Robotics: Nao software documentation, http://www.aldebaran-robotics.com/documentation/
25. Aldebaran Robotics: Nao's homepage, http://www.aldebaran-robotics.com
26. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice Hall Press, Upper Saddle River (2009)
27. Truong-Giang, L.: INI Online (2012), https://sites.google.com/site/inilanguage/
28. Wadhawan, V.: Robots of the future. Resonance 12, 61–78 (2007)