Scripting the swarm: event-based control of microcontroller-based robots.

Stéphane Magnenat[1], Philippe Rétornaz[1], Basilio Noris[2], and Francesco Mondada[1]

[1] LSRO - École Polytechnique Fédérale de Lausanne
stephane at magnenat dot net and firstname.lastname@epfl.ch
http://mobots.epfl.ch
[2] LASA - École Polytechnique Fédérale de Lausanne - firstname.lastname@epfl.ch

Abstract. Swarm robotics in real world requires a large number of robots and thus enough room for experimentation. Therefore, to implement such experiments with limited budget, robots should be compact and low cost, which entails the use of microcontroller-based miniature robots. In this context, developing behaviour is challenging, because microcontrollers are not powerful enough to support common high-level development environments such as Java. Furthermore, the development tools for microcontrollers are not able to monitor and debug groups of robots online. In this paper, we present a new event-based control architecture: ASEBA. It solves the problem of developing and testing collective behaviours by running script inside a lightweight virtual machine on each microcontroller and by providing an integrated development environment to program and monitor the whole group of robots from a single application running on any desktop computer. We have validated ASEBA by implementing a dangerous-area avoidance experiment using the e-puck robot. Experiments of this type are common in swarm robotics, but porting them to real robots is often challenging. By easing the development of complex behaviours on real robots, ASEBA both exposes collective robotics programming to a large community and opens new research perspectives for swarm robotics.

1 Introduction

Swarm robotics is an approach to robotics applications in which a large group of mobile robots coordinate to perform tasks [12]. The core idea is that a lot of relatively simple robots can perform difficult tasks as good as few complex ones. To which extent this claim is verified in reality is still a matter of research [11]. However, it is clear that swarm behaviours require numerous agents [12] and thus enough room for experimentation. These are both expensive, and price is a limiting factor for research. Therefore, robots should be compact and low cost. The class of robots that satisfies at best these constraints are microcontroller-based miniature robots. Our lab has developed a robot of this class, the open-hardware e-puck, which is sold by several companies[3].

[3] e-puck robot: http://www.e-puck.org, which also lists the resellers.

While robots such as the e-puck alleviate the problems of cost and space, developing and testing behaviours is still challenging. On one hand, microcontrollers are not powerful enough to run complete operating systems that would permit the use of standard development tools such as the Java Development Kit. On the other hand, development tools custom-tailored to embedded systems require wired connections with the target, and interact with a single target at a time. Moreover, they usually require to re-flash the whole microcontroller for any single change, and this operation takes. For example, flashing the microcontroller of the e-puck robot lasts one minute. Some swarm-robotics research groups such as DISAL-EPFL have devised a way to flash all the robots at once, but that removes any debugging capability from the development tools. Furthermore, even if each robot is attached to an embedded-system development tool, it is almost impossible to concurrently debug coordinated behaviours implemented in native code. Indeed, codes for the driving of sensors and actuators, for network communication, and for behaviour control are all mixed in a single binary.

To facilitate the development of behaviours on microcontroller-based mobile robots, we have developed a new event-based control architecture: ASEBA, which stands for actuators and sensors event-based architecture (Fig. 1). It solves the problem of developing and testing collective behaviours by running script inside a lightweight virtual machine on each microcontroller and by providing an integrated development environment (IDE) to program and monitor the whole group of robots from a single application running on any desktop computer. The IDE compiles scripts into bytecodes and loads them to the microcontrollers. On the microcontroller of each robot, the virtual machine runs the bytecode. This approach ensures reasonable performances and a good reactivity combined with excellent debugging possibilities. Moreover, in ASEBA, the control code does not poll sensors in a big loop. Instead, the virtual machine executes small parts of the control code that are relevant at the time new information arises, which is the reason why we call ASEBA event-based. For instance, when new data are available from a sensor, the virtual machine calls the data processing code associated with this sensor. If a new messages arrives from another robot, the code that is responsible to react to this information is called as well. This provides a clear understanding of execution flows as asynchronous events provide both intra-robot coordination and inter-robot communication, which helps the development of distributed control algorithms. The ASEBA IDE allows online monitoring of events, which provides aid to the development of the behaviours themselves.

This papers presents ASEBA as a solution to program simple, single-microcontroller mobile robots. It demonstrates its use in a collective robotics experiment using the e-puck robot (Sect. 3), puts it in perspective with the related work (Sect. 4), and discusses its design choices and limitations (Sect. 5). With ASEBA, we address the issue of developing controllers on physical robots; we deliberately do not discuss any development methodology that involves simulation, because the latter has already been studied extensively [10,8].

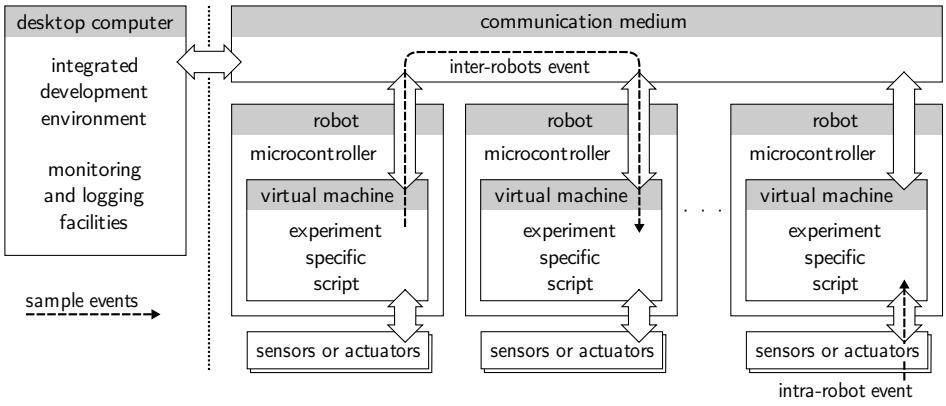

Fig. 1. The overall structure of ASEBA.

2 Aseba

With ASEBA, we write and debug our behaviours from an integrated development environment (IDE) running on a desktop computer (Sect. 2.2). We implement the behaviours using a high-level scripting language (Sect. 2.1). Programming in a scripting language speeds up the writing of behaviours with respect to the traditional use of C in microcontrollers, because developers are free from low-level problems such as memory allocation. The ASEBA IDE compiles script into bytecode and uploads it to the robots microcontrollers. The microcontrollers then execute the bytecode in virtual machines that also provide debugging facilities, such as breakpoints. The use of virtual machines with respect to native code ensures safe execution: no script mistake can bring the microcontrollers into a deadlock state. Moreover, running bytecode inside virtual machines adds flexibility: any 16 bit microcontroller is able to run the bytecode, and thus the ASEBA compiler is not tied to the instruction set of a particular processor. Finally, virtual machines can execute the bytecode from RAM memory, which allows for writing/testing cycles in the order of seconds, where native code must run from flash, which typically takes one minute to update.

In ASEBA, the communication between microcontrollers consists of broadcasted asynchronous messages with payload data, called events. ASEBA uses events both during development — debug commands are events too — and for primary communication during the experiments, typically using wireless radio in collective robotics. An event-based control and communication strategy is interesting for swarm robotics: as radio communication is of limited range, broadcasting events locally allows for scalable controllers. Conversely, master/slave communication is not suitable for a large group of robots, where at a time some may be broken or some may be out of range.

We want to promote cooperation and standardization between researchers using microcontroller-based mobile robots. Thus, we are developing ASEBA as

open source (GPL v.3) and the community can use and modify it free of charge. ASEBA can run on the open-hardware e-puck mobile robot, that has been widely used in research recently. More information as well as the latest version are available at `http://mobots.epfl.ch/aseba.html`.

2.1 Language

In ASEBA, we specify the robots behaviours in an event-based scripting language. An event-based language frees the programmer from managing the moments of execution of code: events trigger the execution of associated code automatically. This is similar to how interrupts act at low-level in microcontrollers.

Syntactically, ASEBA scripts resemble matlab scripts; this similarity allows developers with previous knowledge of some scripting language to feel quickly at ease with robot programming. Semantically, ASEBA script is a simple imperative programming language with a single basic type (16 bits signed integers) and arrays. The use of a simple script allows execution on a compact and lightweight virtual machine that fits well inside a microcontroller. That would not be the case for more complex virtual machines such as the Java Runtime Environment or the .Net Framework. Moreover, this simplicity allows developers to program behaviours with no prior knowledge of a type system; integers being the most natural type of variables and well suited for programming microcontroller-based mobile robots. Furthermore, as ASEBA scripts only permit global variables, the compiler verifies at compile time that neither the stack nor the memory will overflow. This allows to develop robust robotics behaviours. This also reduces the footprint and improves the efficiency of the virtual machine because it performs less checks at run-time than what is required by more dynamic scripts. To perform heavy computations such as signal processing, robots can provide native functions, implemented in C or assembly, that the script can call. Annex A illustrates the use of ASEBA script by showing the complete source code of the experiment we present in Sect. 3.

2.2 Integrated Development Environment

ASEBA eases the development of robots behaviours by providing an IDE within which we edit and debug the scripts for all the robots (Fig. 2). The IDE provides the following features:

- **Concurrent editing**. The IDE provides one tab per robot, with its script, memory content, execution status, and debugging commands. In addition, a toolbar provides general commands that affect all the robots. This allows both an overall control of the group and a specific control of each robot.
- **Syntax highlighting**. The script editor highlights the syntax and colors errors in red. This increases the readability of the scripts.
- **Instant compilation**. The IDE recompiles the script while the developer is typing it. The result of compilation (success or a description of the error) is displayed below the editor. This permits the correction of errors as soon as they appear, which improves the efficiency of the development process.

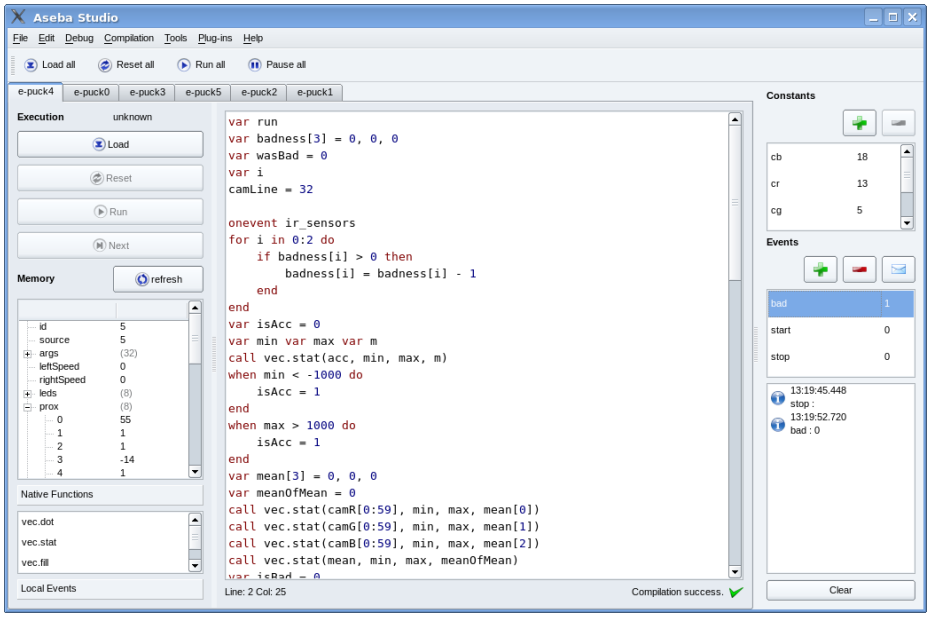

Fig. 2. A screenshot of ASEBA integrated development environment

- **Variables inspection**. The IDE lists the variables available on each robot along with their values, which we can change as well. These variables can represent sensors values, actuators commands, or user-defined variables.
- **Debugger** The IDE integrates a debugger. For each robot, it reports the current execution status. The debugger supports continuous execution, step by step, and breakpoints.
- **Events**. We can specify the names of the events, send them, and monitor the ones that transit over the communication medium. Each robot also provides local events, which are typically emitted when some attached sensors provide updated data.
- **Native functions documentation**. The IDE lists the native functions available on each robot, along with their documentation.
- **Constants definition**. We can define constants that are available to all the robots.

2.3 Virtual Machine

In ASEBA, the robots execute the bytecode in a lightweight virtual machine, with a flash footprint of less than 10 kB and a RAM footprint of 4 kB[4]. The

[4] In our dsPIC30 e-puck implementation, including all communication buffers; this can vary depending on the desired amount of bytecode and variable data, stack size, and number of breakpoints.

virtual machine is implemented in less than 1000 lines of C and embeds the debugging logic. This light footprint is a key element to allow deployment in microcontroller-based robots. The drawback is that the feature set is limited, in particular, our virtual machine does not provide support for object-oriented programming nor for user-defined functions. In comparison, the RAM footprint of leJOS, a simplified Java virtual machine for the Lego Mindstorms, is 17 kB on the RCX brick[5].

3 Sample Experiment

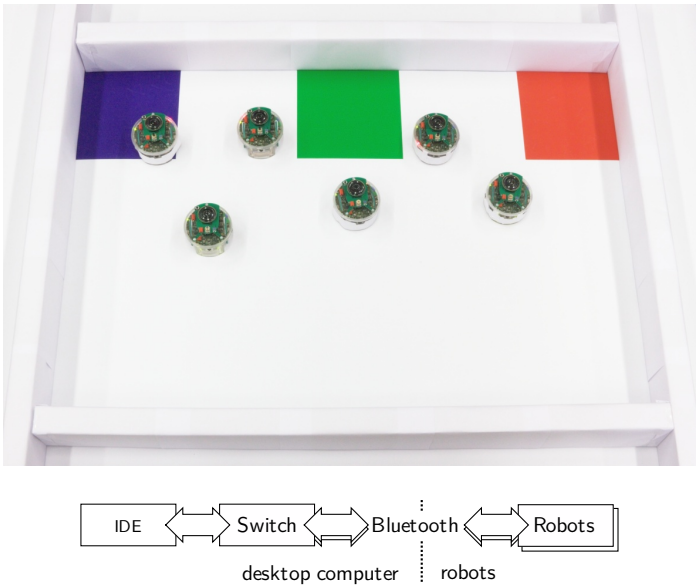

Fig. 3. The experimental setup (top). The structure of communication between the robots and the desktop computer using Bluetooth (bottom).

In this section, we present a simple experiment that demonstrates the use of event-based scripting in collective robotics (Fig. 3). The experiment runs on the open hardware e-puck miniature mobile robot (Fig. 4). A group of e-pucks implements a social behaviour which goal is to avoid dangerous areas. Experiments of this type are common in swarm robotics, because of their interesting dynamics [5]. However, porting them to real robots is often challenging because of the noisy and non-linear nature of real sensors and actuators. ASEBA provides online interactions with the robots sensors and allows dynamic modifications

[5] `http://rcxtools.sourceforge.net/e_lejos.html`

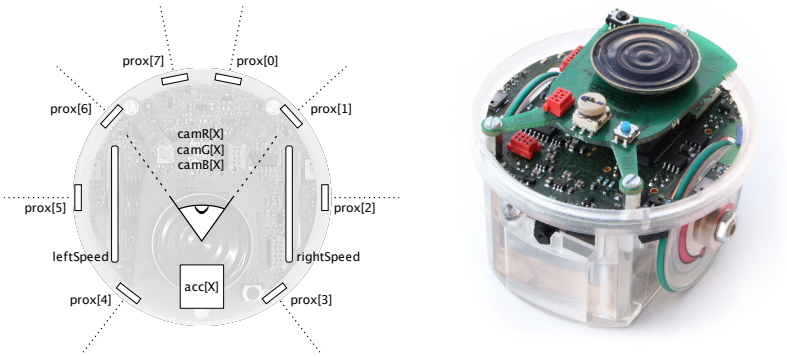

Fig. 4. The e-puck mobile robot and its sensors and actuators as seen by ASEBA: `prox[0:7]` refer to the proximity sensors organized as a ring around the robot; `camR[0:59]/camG[0:59]/camB[0:59]` represent the camera pixels component values; `acc[0:2]` represent the 3-axis accelerometer; `leftSpeed` and `rightSpeed` are the wheel speed commands. A detailed description of the sensors and actuators of the e-puck is available on its web at `http://www.e-puck.org`.

of the robots programs and thus eases the development of behaviours on real robots.

In our sample experiment, the e-pucks perceive the color of the ground with their cameras. If any e-puck experiences a shock while seeing a color, it considers this color as dangerous and transmits this information to the other robots. All the robots avoid the dangerous areas when they see their colors; but with time, they forget the association. The complete source code of the experiment is available in annex (Sect. A); its small size (59 non-empty lines) attests the expressive power of ASEBA in the context of microcontroller-based robots.

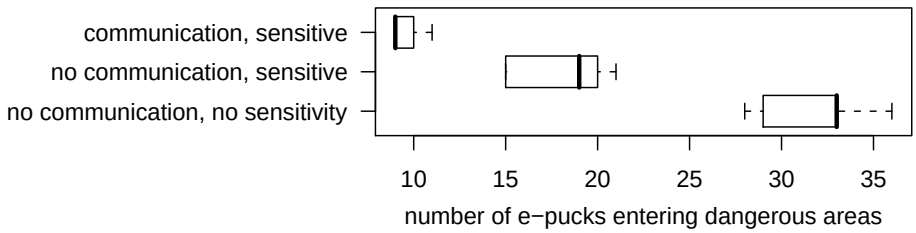

Fig. 5. Results of dangerous area avoidance over 5 runs of 1 minute each; with communication and sensitivity to shocks on and off.

Experimental results (Fig. 5) show that both the perception of external stimuli (sensitivity to shocks) and the use of event-based communication are useful to

avoid dangerous areas. Thanks to the dynamic nature of programming in ASEBA, we were able to implement and run this experiment in one day. This validates that event-based scripting allows the rapid development of swarm behaviours mixing several different sensors. In our case we mix the proximity sensors, the camera, and the accelerometers.

4 Related Work

In this section, we present an overview of the landscape of architectures for developing programs on microcontroller-based mobile robots. We reject solutions that require a desktop-level operating system (such as Linux or Windows). Although these can support environments such as Java, they require more expensive hardware than microcontrollers and are thus not suitable for experiments involving large groups of robots. Table 1 shows a comparison of the architectures we present.

	IDE	Debug	Easy	Language	E. b.	Emb.	HW i.	Cross-p.	Open-s.
ASEBA	Yes	Yes	Yes	Specific	Yes	Yes	Yes	Yes	Yes
Mindstorm	Yes	No	Yes	Any	No	Yes	No	No	Yes
Player	No	No	No	Any	No	No	Yes	Partial	Yes
Urbi	No	No	Yes	Specific	No	No	Yes	Yes	Yes
Pyro	No	Yes	Yes	Python	No	No	Yes	Yes	Yes
Cubesystem	No	No	No	C/C++	No	Yes	No	Yes	No
Matlab/LabView	Yes	Yes	Yes	Specific	No	No	Yes	Yes	No
IDE for embedded	Yes	Yes	No	C/C++	n/a	Yes	No	Yes	No

Table 1. Functionality comparison of architectures for microcontroller-based mobile robots. IDE means that an integrated development environment is available, with or without a debugger. *Debug* means that a debugger is available. *Easy* means that the language is simple enough to be used by non-experts. *E. b.* means event-based, it indicates whether the control structure is based around asynchronous events or not. *Emb.* indicates whether the user program can reside in the microcontroller of the robot itself. *HW i.* denotes whether the architecture is tied to a specific robot hardware. *Cross-p.* and *Open-s.* indicate whether the programming environment and the execution runtime are cross-platform and open-source.

– **The Lego Mindstorms**. The Lego Mindstorms NXT is one of the easiest way to build and prototype physical robots [7]. The core of the NXT uses a powerful 32bit ARM7 microcontroller with 64 kB of RAM, can connect to up 3 servo motors and 4 sensors, and is capable of Bluetooth communication. Its sensors set comprises touch, sound, light, and ultrasonic sensors.

The Lego Mindstorms robot can be assembled from any choice of Lego bricks which allows to build various robots quickly. From the software standpoint, a node-based graphical interface allows to design behaviours (e.g. by connecting sensor inputs to conditional blocks and motor activation). Additionally, the Not eXactly C (NXC) provides a concurrent C-like language that runs directly on the NXT brick and benefits from a strong community support in terms of examples, tutorials, and tools. One can also control the NXT from Matlab or Simulink. Finally, a simplified Java virtual machine runs on the NXT[6], which is possible because the NXT has a large amount of RAM for a microcontroller (64 kB vs 8 kB on the e-puck). Nevertheless, all the development tools for the NXT platform lack a debugger and the platform itself lacks the communication and the deployment capabilities that would make it suited to collective robotics.

- **Player** [6]. Player is one of the most widely used software platform for robots programming. It provides a suite of tools to interface the sensors and actuators of a robot with a client program running on an external computer. On this computer, Player provides an interface for a large number of programming languages. However, with Player the controller does not run on the robot itself and thus is subject to delay, bandwidth limitation, and scalability problems due to the communication.
- **URBI** [1]. URBI is a parallel, event-driven scripting language with an interface to C++ objects. In the context of robotics, it provides a powerful way to quickly create complex behaviours. URBI supports several robots (e.g. Aibo, iRobot Create, Lego Mindstorms NXT). Unfortunately, URBI programs do not run directly on the robot and thus URBI is subject to the same limitations as Player.
- **Pyro** [3]. Pyro is a Python based programming framework focusing on education. Pyro uses a client/server architecture with a minimal server on the robot itself and the main program running on an external machine. Pyro can interface with Player (and thus with all robots compatible with it) or directly (among others) with the Pioneer, Khepera, AIBO, and Roomba robots. Pyro suffers from the same drawbacks as Player do.
- **CubeSystem** [2]. CubeSystem is a combination of hardware and software components that can be assembled to create a complete system. The CubeSystem revolves around the RoboCube (an embedded controller), the CubeOS operating system, and the RoboLib which contains a collection of functions for robotics. It provides drivers for a large number of sensors and actuators, and interfaces with a high-level language for programming behaviours, such as PDL. CubeSystem is limited to the RoboCube embedded computer and is not open source, which prevents its use in custom-made robots.
- **Matlab / LabView**. Mathwork Matlab [7] and National Instrument LabView [8] are both development platforms with a powerful engine for computation and a relatively simple programming language. They can interface

[6] leJOS, Java for LEGO Mindstorms: `http://lejos.sourceforge.net`
[7] MathWorks Matlab: `http://www.mathworks.com`
[8] National Instruments LabView: `http://www.ni.com/labview`

with robots natively or through external libraries or drivers, depending on the hardware. Furthermore, they also provide a graphical user interface and a debugging environment. LabView also provides node-based programming (Lego Mindstorms NXT uses LabView). However, they are not free, and they do not allow the creation and debug of a standalone application that runs on a microcontroller.

- **IDE for embedded systems**. Almost any microcontroller is supported by one or more IDE, either from its vendor or from third party. These IDE typically compile C and assembly and provide a debugger, and often a simulator. However, they are general tools with no particular support for robotics applications and they typically require a wired connection with the microcontroller.
- **Robot programming libraries**. Open source libraries, such as YARP [9], Orocos [4] and many others, provide clean modular architectures for robot programming. In most cases they provide a solid communication interface for message and object passing, access to common input/output devices (such as cameras, motor control boards, etc.) and can be extended to support any type of hardware. However, most of these libraries are client/server oriented and run on external or embedded computers, and thus suffer from the same limitations as Player do.

5 Discussion

Beside ASEBA, there is no development tool for miniature microcontroller-based robots that at the same time run code on the robots themselves, are event-based, and provide a high-level scripting language and an integrated development environment (Table 1).

Running the controller on the robot — with opposition to running it on a remote computer — is critical for scalability. Indeed, it is not possible to deploy a large group of remote-controlled robots because this would require too many communication links. Moreover, remote control adds latency between perception and action which is a limiting factor from a control perspective.

An event-based control and communication strategy is interesting for collective robotics. The ASEBA implementation on the e-puck, which uses Bluetooth, requires global communication; but ASEBA as an architecture supports local communication, for instance radio links of limited range. In such cases, broadcasting events locally allows for scalable controllers. In contrary, master/slave communication is not suitable for a large group of robots, where at a time some may be broken or some may be out of range. ASEBA pushes the concept further by unifying events for internal sources and events from other robots. This unification provides a coherent control structure where events are the sole source of code execution. This removes the classical polling loops found in other robots controllers which simplifies the real-time aspect of robots programming.

Scripting mobile robots allows to reduce the time from the experiment design to the working behaviour. Indeed, script is faster to write and safer to run than

C code. This ensures a low entry curve for new programmers and permits to do more research or development for the same amount of resources. Scripting is known for its low performances with respect to native code, in particular when running on simple virtual machines, which is the case in ASEBA. However, as microcontrollers can provide highly optimized native functions, for instance ones that do exploit the optimized instructions of the microcontrollers. The limitations of ASEBA script, such as lack of local variable, are the result of an informed choice: Should we have added more dynamics into the language, would the required virtual machines have became slower and heavier.

An integrated development environment with a debugger is an important, yet often neglected, element for efficient development of computer programs. While this fact is well known in computer science, the current state of the art of collective robotics, in particular with miniature microcontroller-based robots, seldom provides such tool. Embedded systems IDE do exist, but are based on wired connections. They target single fixed systems, not swarms of robots. For instance, it is not realistic to connect six e-pucks to six Microchip MPlab IDE with six cables and expect the robots to move and act as if they were not being debugged. ASEBA demonstrates that with the appropriate architecture, it is possible to develop and debug robots in real conditions. Online monitoring and logging of events, because of the close link between events and the behaviour of the swarm, improves dramatically the understanding of what the swarm is doing at each moment. This capability enhances the classical debugging facilities such as breakpoints and facilitates the development of complex behaviours.

ASEBA on the e-puck is not perfect and suffers from several limitations. First, communication is based on the rfcomm protocol of Bluetooth, which only allows up to 7 concurrent communications. However, this is a limitation of the current e-puck implementation and not of the core concept: the use of a lower level radio communication medium would remove this limitation. Second, while debugging, the interaction between the IDE and the robots brings a communication bandwidth consumption overhead that is linear with the number of robots, which limits the scalability. However, ASEBA supports the storage of bytecode inside the flash memory of the robots microcontrollers, which allows the deployment and execution of collective control code on a large group of robots, even if the development was limited to only a subset of them. These limitations bear some resemblance with the ones of embedded systems IDEs; however the use of wireless communication and lightweight debug protocol makes ASEBA far more powerful.

Being portable and open source, ASEBA is adaptable to multiple targets. In particular, it is readily available on the open-hardware e-puck mobile robot.

6 Conclusion

ASEBA provides a consistent solution to the development and test of behaviours for swarms of small microcontroller-based robots. By easing the development of complex behaviours on real robots, ASEBA both exposes collective robotics

programming to a large community and opens new research perspectives for swarm robotics.

7 Acknowledgments

We thank Cyrille Dunant and an anonymous reviewer for their feedback on the draft of this paper.

This work was supported by the Swarmanoid and the Perplexus projects, which are funded by the Future and Emerging Technologies program (IST-FET) of the European Community. The information provided is the sole responsibility of the authors and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication.

References

1. J.-C. Baillie. Urbi: towards a universal robotic low-level programming language. In *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 820–825, 2005.
2. A. Birk. Fast robot prototyping with the cubesystem. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*, pages 5177–5182, 2004.
3. D. Blank, D. Kumar, M. L., and H. Yanco. Pyro: A python-based versatile programming environment for teaching robotics. *Journal of Educational Resources in Computing (JERIC)*, 3(4), 2003.
4. H. Bruyninckx. Open robot control software: the orocos project. In *Proceedings of the 2001 IEEE International Conference on Robotics and Automation*, pages 2523–2528, 2001.
5. D. Floreano, S. Mitri, S. Magnenat, and L. Keller. Evolutionary Conditions for the Emergence of Communication in Robots. *Current Biology*, 17:514–519, 2007.
6. B. P. Gerkey, R. T. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the Intl. Conf. on Advanced Robotics (ICARJ)*, pages 317–323, 2003.
7. F. Klassner. A case study of lego mindstorms'^{TM}suitability for artificial intelligence and robotics courses at the college level. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 8–12. ACM, 2002.
8. A. Martinoli, K. Easton, and W. Agassounon. Modeling Swarm Robotic Systems: A Case Study in Collaborative Distributed Manipulation. *Int. Journal of Robotics Research*, 23(4):415–436, 2004.
9. G. Metta, P. Fitzpatrick, and L. Natale. Yarp: Yet another robot platform. *Advanced Robotics Systems*, 3:43–48, 2006.
10. O. Michel. Webots: Symbiosis between virtual and real mobile robots. In *Virtual Worlds*, pages 254–263. Springer, 1998.
11. F. Mondada, G. C. Pettinaro, A. Guignard, I. V. Kwee, D. Floreano, J.-L. Deneubourg, S. Nolfi, L. M. Gambardella, and M. Dorigo. SWARM-BOT: A new distributed robotic concept. *Autonomous Robots*, 17(2–3):193–221, 2004.
12. E. Şahin. Swarm robotics: From sources of inspiration to domains of application. In *Swarm Robotics*, pages 10–20. Springer, 2005.

A Complete Source Code of Experiment

The code starts by defining some global variables (part A). Then, the code associated with the intra-robot infrared sensors event implements the actual controller. First, the controller decreases the values of the association of bad colors; it uses a `for` loop to iterate over the colors (part B). Then, it checks if there is a shock by looking at the values of the accelerometers. It first calls the `vec.stat` native function to get the minimum and the maximum of the acceleration on all axis, and then uses a `when` conditional to react only if the conditions are different than during the previous execution (part C). Next, the controller processes raw colors from the camera by extracting their means (part D). Then, if the robot sees a color and is experiencing any shock, it sends an event to notify the other robots (part E). At the end of the event, the controller sets the speed of the wheels in order to avoid bad colors (part F). When a robot receives an event telling it that a color is bad, the association is set (part G).

```
var badness[3] = 0, 0, 0                                    # part A
var wasBad = 0
var i
camLine = 32

onevent ir_sensors                                          # part B
for i in 0:2 do
    if badness[i] > 0 then
        badness[i] = badness[i] - 1
    end
end

var isAcc = 0                                               # part C
var min var max var m
call vec.stat(acc, min, max, m)
when min < -1000 or max > 1000 do
    isAcc = 1
end

var mean[3] = 0, 0, 0                                       # part D
var meanOfMean = 0
call vec.stat(camR[0:59], min, max, mean[0])
call vec.stat(camG[0:59], min, max, mean[1])
call vec.stat(camB[0:59], min, max, mean[2])
call vec.stat(mean, min, max, meanOfMean)
var isBad = 0

var activeColor = 3                                         # part E
if meanOfMean > 38 then
    activeColor = 2
    if mean[0] > mean[1] and mean[0] > mean[2] then
        activeColor = 0
    elseif mean[1] > mean[2] and mean[1] > mean[0] then
```

```
        activeColor = 1
    end
    if isAcc == 1 then
        emit bad activeColor
        badness[activeColor] = 1200
    end
end
if activeColor != 3 then
    if badness[activeColor] > 0 then
        isBad = 1
    end
end

if wasBad == 1 then                                    # part F
    if prox[1] + prox[6] + prox[0] + prox[7] > 300 then
        leftSpeed = -400
        rightSpeed = 400
    else
        wasBad = 0
    end
else
    if isBad == 0 then
        leftSpeed = 1000 - prox[1] * 4 - prox[0] * 2
        rightSpeed = 1000 - prox[6] * 4 - prox[7] * 2
    else
        leftSpeed = -400
        rightSpeed = 400
        wasBad = 1
    end
end

onevent bad                                            # part G
badness[args[0]] = 1200
```