ASEBA Meets D-Bus: From the Depths of a Low-Level Event-Based Architecture into the Middleware Realm

Stéphane Magnenat and Francesco Mondada

Abstract— **The robotics research community has clearly acknowledged the need of open and standard software stacks to promote reuse of code and developments. However, to date no particular project has prevailed. We suggest that one possible reason for this is that most middleware do not address issues specific to robotics, such as writing, monitoring, and debugging real-time behaviors close to hardware. In this light, we present ASEBA, an event-based architecture for mobile robots with microcontrollers and a Linux board. In these, the microcontrollers manage sensors and actuators locally and the Linux board runs the high-level control. The popularity of these robots and the role of microcontrollers are increasing. ASEBA achieves vertical integration by bringing the facilities of scripting inside the microcontrollers and by bridging them with programs running on Linux. To program the microcontrollers, ASEBA provides an integrated development environment. The latter compiles a simple scripting language into bytecode which runs in the virtual machines.**

We demonstrate a robot remote control application where low-level scripts prevent collisions. At the Linux level, this application employs both Perl and Python programs which communicate with ASEBA through D-Bus (D-Bus is a middleware present by default under Linux). This application shows how convenient it is to program all parts of the robot thanks the vertical integration of ASEBA.

We think that because it considers the needs of robotics software development at all levels, the integrative approach of ASEBA might be a way to overcome the stall in standardization.

I. INTRODUCTION

The robotics research community has clearly acknowledged the need of open and standard software stacks to promote reuse of code and developments between researchers [1]. To that end, various researchers have proposed a wide range of software *middlewares* [2]; however to date none has prevailed. We suggest that one possible reason for this is that most middleware do not address issues specific to robotics. Indeed, a middleware proposes abstractions from a computer science point of view [3]; yet the range of problems that a robotics software stack must deal with is much wider. In particular, a complete and comprehensive stack should allow writing, monitoring, and debugging real-time behaviors close to hardware.

This work was supported by the Swarmanoid project, funded by the Future and Emerging Technologies programme (IST-FET) of the European Community, under grant IST-022888. 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. We thank Emmanuel Eckard for proofreading the manuscript and Basilio Noris for the photo of the handbot.

Stéphane Magnenat and Francesco Mondada are with EPFL-LSRO, Station 9, 1015 Lausanne, Switzerland - `http://mobots.epfl.ch`. Please send correspondence to `stephane at magnenat dot net`.

In this paper, we present ASEBA in this light. ASEBA is an event-based architecture that runs on miniature multi-microcontrollers mobile robots [4], [5]. By providing a clean and unified interface to the robot hardware and by running user code inside a virtual machine (VM) on the microcontrollers, ASEBA brings the flexibility and reusability of middleware deep into the robots' mechatronic. Moreover, ASEBA seamlessly integrates with Linux programs through D-Bus, the standard messaging middleware on modern Linux distributions [6].

ASEBA is based on the experience we gained by developing the s-bot mobile robot [7]. The s-bot was a complex and highly integrated mobile robot that embedded a number of sensors and actuators. It was based around an ARM Linux board which connected to PIC microcontrollers through an I^2C bus. This structure of dumb microcontrollers all polled from an intelligent Linux board is the architectural assumption of most middlewares. The s-bot suffered from overload of the I^2C bus, because the control code running on Linux was polling the sensors and updating the actuators at regular intervals. For our new mobile robots (Fig. 1 and Fig. 2), we have decided to move from a polling to an event-based architecture. To achieve this we have kept the same type of hardware structure, but we have replaced the I^2C bus with a CAN bus which is capable of asynchronous communication [8]. We have also replaced the PIC microcontrollers with dsPICs, which are faster and provide a larger memory. This has allowed to move part of the control code into the microcontrollers and let them communicate through events. They can now filter raw data and implement pre-processing close to the data sources, which offloads the bus and the Linux computer.

To mediate the low-level (inside microcontrollers, for instance obstacle avoidance) and the high-level (inside Linux, for instance vision) realms of the robot behavior, ASEBA provides a software hub called *Medulla*. Medulla allows any Linux software to access the events and the data of the microcontrollers through D-Bus. This paper presents the implementation of ASEBA, describes its integration into Linux using Medulla, and demonstrates a complete robot controller.

II. ASEBA

A. Architecture

ASEBA is an event-based architecture consisting of a network of processing units which communicate using asynchronous events. An event is a message with an identifier and payload data. All nodes send events and react to incoming events. In miniature mobile robots, a typical network contains several microcontrollers and a Linux board which

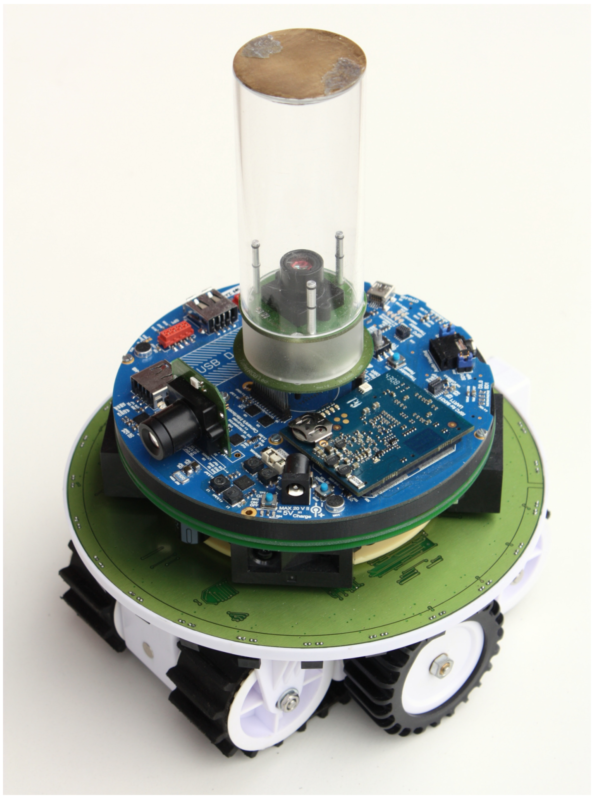


Fig. 1: The marXbot robot.

communicate through a CAN bus. The Linux runs a piece of software, Medulla, which extends the network through D-Bus to local programs and through TCP-IP to any remote host. This allows to run high-level controllers on the Linux and to connect an integrated development environment (IDE) from a desktop computer (Fig. 3a).

Programming microcontrollers is traditionally a time-consuming task restricted to specialists. To provide flexibility and accessibility while distributing processing to the microcontrollers, ASEBA runs the user code inside a tiny VM (Fig. 3b in invert video). The user code consists of bytecode that is compiled out of a simple event-based scripting language, which syntactically resembles matlab scripts (see Listing 1). Semantically, this language is of imperative style with arrays of 16 bit signed integers as the only data type. In this language, receiving an event executes the associated part of the script, if any. Events can originate from an internal peripheral of the microcontroller, from a Linux program, or from another microcontroller. This association frees the programmer from managing the timing of code execution (see Sec. II-B). To structure the code, the programmer can define subroutines which can be called from any subsequent code. In addition to the usual *if* conditional, the ASEBA language provides the *when* conditional which is true when the actual evaluation of the condition is true and the last was false. This allows the execution of a specific behavior when a state changes, for instance when an obstacle is closer than a threshold distance. The VM exports sensors values and actuators commands as

script of proximity sensors microcontroller:

```
var vectorX[24] = -254, -241, ...
var vectorY[24] = -17, -82, ...
var targets[2]
var eventBuffer[2]
var activation

sensors.period = 50

onevent SetSpeed
targets[0] = event.args[1] + event.args[0]
targets[1] = event.args[1] - event.args[0]

onevent sensors.updated
call math.dot(eventBuffer[0], proximity.corrected,
  vectorX, 15)
call math.dot(eventBuffer[1], proximity.corrected,
  vectorY, 15)
call math.dot(activation,eventBuffer,eventBuffer,0)

if activation > 600 then
  emit ObstacleDetected eventBuffer
end

when activation <= 600 do
  emit FreeOfObstacle
end<
```

script of left track microcontroller:

```
var user_target = 0
var obstacle_target = 0

sub UpdateTargetSpeed
motor.pid.target_speed=user_target+obstacle_target

onevent SetSpeed
user_target = event.args[0]
callsub UpdateTargetSpeed

onevent ObstacleDetected
obstacle_target = event.args[0] + event.args[1]
callsub UpdateTargetSpeed

onevent FreeOfObstacle
obstacle_target = 0
callsub UpdateTargetSpeed
```

script of right track microcontroller:

```
var user_target = 0
var obstacle_target = 0

sub UpdateTargetSpeed
motor.pid.target_speed=user_target+obstacle_target

onevent SetSpeed
user_target = event.args[1]
callsub UpdateTargetSpeed

onevent ObstacleDetected
obstacle_target = event.args[0] - event.args[1]
callsub UpdateTargetSpeed

onevent FreeOfObstacle
obstacle_target = 0
callsub UpdateTargetSpeed
```

Listing 1: Example of ASEBA scripts distributed among three microcontrollers. These scripts implement obstacle avoidance on a marXbot robot (Fig. 1) using potential fields. The corresponding ASEBA network is shown in Fig. 4.

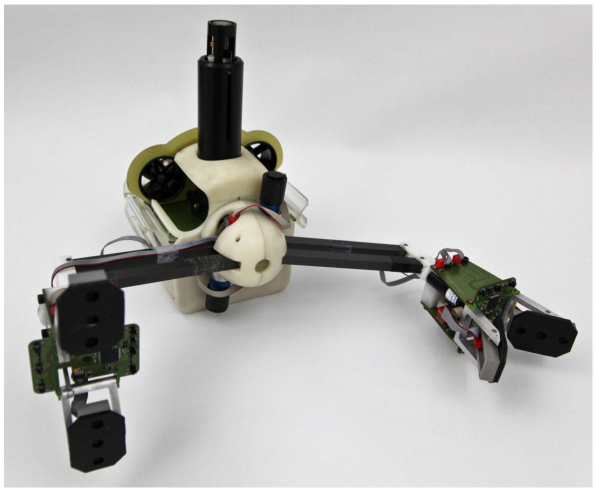
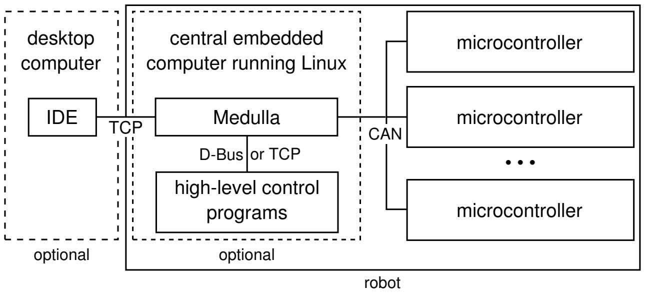
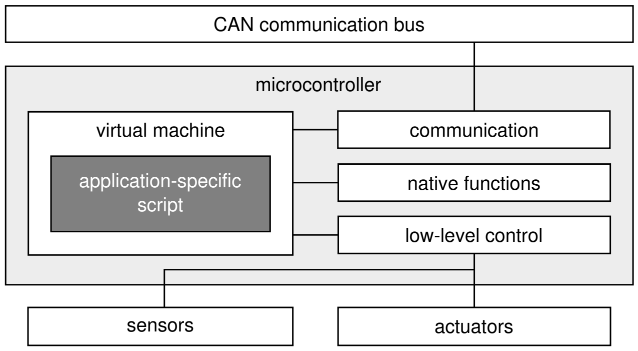


Fig. 2: The handbot robot.



(a) a typical ASEBA network in a robot



(b) a microcontroller in an ASEBA network

Fig. 3: ASEBA in miniature mobile robots.

normal variables, which enables a seamless access to the hardware.

When programming complex robots containing a lot of sensors and actuators, it is important to be able to inspect the value of a variable in live or to monitor events. ASEBA provides an IDE which unifies all the development and debugging process in a single application (Fig. 5). It shows the state of all the VM and logs all the events transiting over the network. Moreover, it can plot a graph of the events' values over time. For each microcontroller, the IDE shows and allows the edition of the values of the sensors, the actuators, and the user-defined variables. The IDE provides a script editor

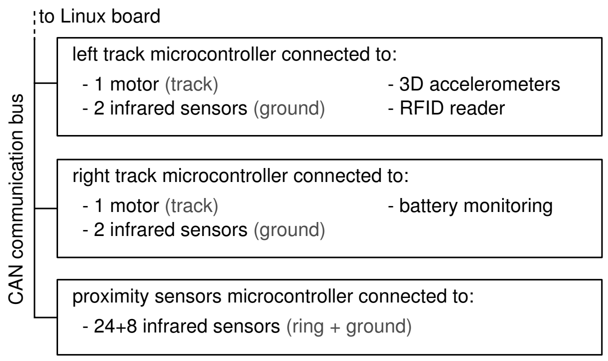
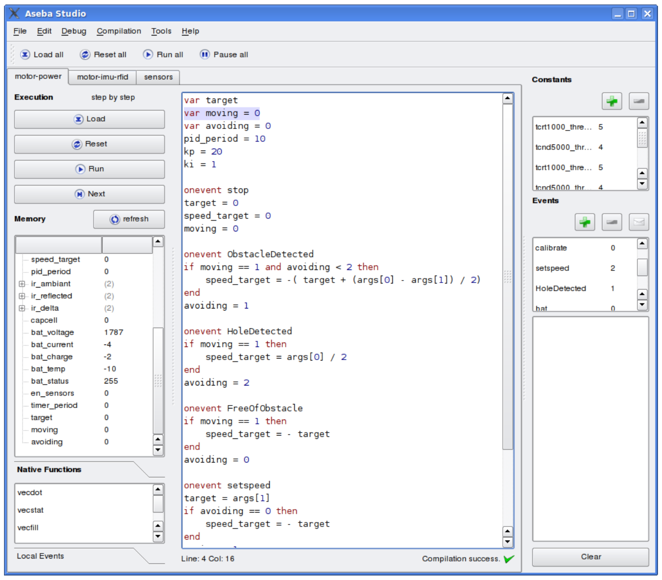

to Linux board
CAN communication bus
left track microcontroller connected to:
- 1 motor (track)
- 2 infrared sensors (ground)
- 3D accelerometers
- RFID reader
right track microcontroller connected to:
- 1 motor (track)
- 2 infrared sensors (ground)
- battery monitoring
proximity sensors microcontroller connected to:
- 24+8 infrared sensors (ring + ground)

Fig. 4: Structure of the ASEBA network of the script in Listing 1.



speed_target	0
pid_period	0
ir_ambiant	(2)
ir_reflected	(2)
ir_delta	(2)
capcell	0
bat_voltage	1787
bat_current	-4
bat_charge	-2
bat_temp	-10
bat_status	255
en_sensors	0
timer_period	0
target	0
moving	0
avoiding	0

vecdot
vecstat
vecfill

```
var target
var moving = 0
var avoiding = 0
pid_period = 10
kp = 20
ki = 1

onevent stop
target = 0
speed_target = 0
moving = 0

onevent ObstacleDetected
if moving == 1 and avoiding < 2 then
    speed_target = -( target + (args[0] - args[1]) / 2)
end
avoiding = 1

onevent HoleDetected
if moving == 1 then
    speed_target = args[0] / 2
end
avoiding = 2

onevent FreeOfObstacle
if moving == 1 then
    speed_target = - target
end
avoiding = 0

onevent setspeed
target = args[1]
if avoiding == 0 then
    speed_target = - target
end
```

Line: 4 Col: 16   Compilation success.

tcrt1000_thre...	5
tcnd5000_thr...	4
tcrt1000_thre...	5
tcnd5000_thr...	4

calibrate	0
setspeed	2
HoleDetected	1
bat	0

Fig. 5: Screenshot of ASEBA IDE.

with syntax highlighting. This editor compiles script while the programmer is typing it and visually marks errors. If the script is free of compilation error, the programmer can upload it to and run it on the microcontroller in two clicks. A distributed debugger let the programmer set breakpoints and control the execution state of each microcontroller, for instance to do step by step inside the script. If a script performs an illegal operation, such as a division by zero, its execution is halted and the faulty line is highlighted.

Bytecode running inside a VM is slower than well-written native code. To allow relatively heavy processing such as vector arithmetic, ASEBA provides libraries of native functions. A standard library available on each microcontroller provides vector operations and trigonometric functions; specific functions are also available depending on the microcontroller.

The ASEBA compiler runs in the IDE or in Medulla. It consists of a top-down hand-written parser which produces an abstract syntax tree. A type checker verifies this tree, and an optimizer performs simple improvements such as dead-code elimination, constant array access elimination, etc.

addresses (in 16 bit words)	content

`bytecodeSize −1`
`...`
	unused bytecode

`...`
`evLastAddr`
	bytecode for last managed event

	...

`...`
`ev0Addr`
	bytecode for first managed event

`evVectSize −1`	`evLastAddr`
`evVectSize −2`	`evLastId`
`...`	`...`
`0x0002`	`ev0Addr`
`0x0001`	`ev0Id`
`0x0000`	`evVectSize`

- `evVectSize` is the size of events vector.
- `ev0Addr` is the starting address of first managed event.
- `ev0Id` is the identifier of first managed event.
- `evLastAddr` is the starting address of last managed event.
- `evLastId` is the identifier of last managed event.
- `bytecodeSize` is the total number of bytecode words.

TABLE I: The program memory layout of an ASEBA VM.

addresses (in 16 bit words)	content

`variablesSize −1`
`...`
	temporary variables to pass constants to native calls

`...`	unused variables

`...`
`exportedVarsLength`
	user-defined variables

`...`
`0x0000`
	exported variables

- `exportedVarsLength` is the length of the exported variables,
- `variablesSize` is the total number of variable words.

TABLE II: The data memory layout of an ASEBA VM.

The resulting simplified tree is transformed into bytecodes corresponding to each event, which are linked together in the final bytecode.

ASEBA is written in C (microcontrollers) and C++/Qt (IDE). It is open-source (GPL v.3) and fully cross-platform. More information, as well as the latest version, are available at `http://mobots.epfl.ch/aseba.html`.

B. Virtual Machine Implementation

The VM of ASEBA implements a Harvard architecture and performs computations as a stack machine. Its state thus consists of program memory (TABLE I), data memory (TABLE II), stack memory, program counter, flags, and the list of breakpoints.

The VM implements events as a physical processor would implement interrupts. In the bottom of the program memory, a table called *events vector* maps events' identifiers with addresses. When an event corresponding to an entry arrives, the VM executes the corresponding code until it reaches a *stop* bytecode or it has executed too many steps.

Each bytecode consists of one or more 16 bit words. In the first word, the 4 most significant bits encode the bytecode's type; the rest and the following words encode the bytecode's data. The execution of a bytecode increments the program counter by its words count, excepted for bytecodes performing flow control which jump otherwhere. TABLE III shows all types of bytecodes. The bytecode can be flashed into the microcontrollers, so a network of microcontrollers can run autonomously.

name	w.c.	function
stop	1	stop execution
small immediate	1	push a constant onto the stack
large immediate	2	push a constant onto the stack
load	1	push data from memory onto the stack
store	1	pop data from the stack into the memory
load indirect	2	push data from memory onto the stack using an offset from the stack
store indirect	2	pop data from the stack into the memory using an offset from the stack
unary arithmetic	1	unary arithmetic operation on the stack
binary arithmetic	1	binary arithmetic operation on the stack
jump	1	jump to another execution address
conditional branch	2	check a condition on the stack and jump depending on the result
emit	3	send an event
native call	1	call a native function
sub call	1	jump into a subroutine, store return adress on the stack
sub ret	1	return from a subroutine, using return address from the stack

TABLE III: The types of ASEBA bytecodes. The w.c. column indicates the number of words the type of bytecode counts.

The bottom of the data memory contains the exported variables, whose names and meanings are pre-defined per microcontroller. These include the identifier of the microcontroller and the payload data of the last event, but also all the variables exported by the sensors and the actuators.

The VM is a lightweight software. In a typical dsPIC33 implementation, it consumes 10 kB of flash memory and 4 kB of RAM, including all communication buffers. We can adapt these requirements by adjusting the amount of bytecode and variable data, stack size, and number of breakpoints.

C. Medulla, the D-Bus Integration

Medulla presents the ASEBA network through a singleton object of interface `ch.epfl.mobots.AsebaNetwork` (Listing 2). Through this interface, any program can retrieve information about the network, read and write variables, load scripts into the microcontrollers, or send events. Receiving events requires a bit more machinery, for the sake of efficiency. Indeed, in a multi-application context, each program is only interested in some events. To prevent waking-up every program for each event, Medulla implements events filtering. Each application that wants to receive events must call `CreatEventFilter()` to create an event filter. The latter exports the interface `ch.epfl.mobots.EventFilter`, which allows the application to choose which events it wants to receive. The application will then receive events through the `Event` signal.

III. Example of Application

This section presents a remote control application for the marXbot robot. It illustrates the vertical integration

```
interface ch.epfl.mobots.EventFilter
{
  method Void ListenEvent(UInt16 eventId)
  method Void ListenEventName(String eventName)
  method Void IgnoreEvent(UInt16 eventId)
  method Void IgnoreEventName(String eventName)
  signal Event(UInt16 id, String name,
    Array<SInt16> payloadData)
}

interface ch.epfl.mobots.AsebaNetwork
{
  method Void LoadScripts(String fileName)
  method Array<String> GetNodesList()
  method Array<String> GetVariablesList(
    String nodeName)
  method Void SetVariable(String nodeName,
    String variableName, Array<SInt16> variableData)
  method Array<SInt16> GetVariable(String nodeName,
    String variableName)
  method Void SendEvent(UInt16 eventId,
    Array<SInt16> payloadData)
  method Void SendEventName(String eventName,
    Array<SInt16> payloadData)
  method ObjectPath CreateEventFilter()
}
```

Listing 2: The D-Bus interface of the ASEBA network, as exported by Medulla.

```
#!/usr/bin/perl
use Net::DBus;
use Net::DBus::Reactor;

# gets stub of ASEBA network
my $bus = Net::DBus->session;
my $asebaService = $bus->get_service(
  'ch.epfl.mobots.Aseba');
my $asebaNetwork = $asebaService->get_object('/',
  'ch.epfl.mobots.AsebaNetwork');

# loads scripts
$asebaNetwork->LoadScripts($ARGV[0]);

# creates an event filter and listen for an event
my $eventFilterPath =
  $asebaNetwork->CreateEventFilter();
my $eventFilter = Net::DBus::RemoteObject->new(
  $asebaService, $eventFilterPath,
  'ch.epfl.mobots.EventFilter');
$eventFilter->ListenEventName('SetSpeed');
$eventFilter->connect_to_signal('Event',
  sub {
    # print the event
    my ($id, $name, $payloadData) = @_;
    print 'Event␣' . $id . '/'. $name . ':␣';
    for my $value (@$payloadData) {
      print "$value␣";
    }
    print "\n";
  }
);

# starts event loop
my $reactor = Net::DBus::Reactor->main();
$reactor->run();
exit(0);
```

Listing 3: A Perl program to load an ASEBA script and to log an event. This program uses the `libnet-dbus-perl` library.

that ASEBA allows. This applications uses three events: `SetSpeed` allows a human to control the robot's movements, while `ObstacleDetected` and `FreeOfObstacle` allow the robot to avoid obstacles regardless of the human's control command. At the low level, distributed among three microcontrollers, three ASEBA scripts implement the obstacle avoidance and its fusion with the control command (Listing 1). At the Linux level, a Perl program loads this script, and then dumps all the `SetSpeed` events using an event filter (Listing 3). A Python program sends `SetSpeed` events to the microcontrollers at regular intervals, if the joystick has a new position (Listing 4). This application shows that thanks to its VM and Medulla, ASEBA allows a straightforward scripting of the robot's behaviors at all levels.

IV. Related Work and Discussion

There are multiple middlewares that provide event-based communication. They mostly differ by the communication layer they use. Some use well-known protocols such as HTTP [9] or CORBA [10], [11]; and some provide their own [12], [13]. However, they all exhibit the same basic structure: a software architecture where components interact through a communication layer. They do not provide any feature to ease the interaction with the robot's hardware managed by microcontrollers. The idea of distributing processing over multiple sensors has been explored in theory almost twenty years ago [14]. However, this work focused on a mathematical model and did not propose any implementation. In mobile robots, several works have explored using the multi-master capabilities of the CAN bus to asynchronously transfer data. They have proposed that the sensors send data at a pre-defined [15] or adaptive [16] rate. The idea of using a VM to bring flexibility to microcontrollers is not new either [17]. However, to our knowledge ASEBA is the first vertical integration of a software stack from the microcontrollers to the Linux applications.

We consider that the major limitation of ASEBA is its single data type, which is basically 16 bit integers organized in arrays. While this allows the embedding of the VM into most of the existing microcontrollers, it limits the types of data that Linux programs can exchange through ASEBA. If ASEBA implemented the same set of data structures as D-Bus does, we could further hide the difference between microcontrollers and Linux programs. This would correspond to strongly typing events as in [18]. However, we must keep in mind that much of the ease of programming in ASEBA comes from the static memory allocation of data inside the VM. The compiler knows the address of each variable globally, which allows it to perform useful checks at compile time. If we want richer data structures, we would reduce the user-friendliness of the environment. We could alleviate this drawback by adding run-time checks and by improving the reasoning done by the compiler. While the former would reduce the execution speed and increase the bytecode size, the second is promising but requires state of the art techniques from research in compilers.

V. Conclusion

ASEBA allows vertical integration between the various software layers of a modern, multi-microcontrollers robot. At the level of the microcontrollers, ASEBA takes profit of the closeness to hardware to filter raw data and implement reflex-like control locally. With respect to polling the microcontrollers, this allows lower latency reactions and a reduced load for the Linux computer. Thanks to the easy to use scripting language and the IDE, ASEBA brings these advantages without compromising the flexibility nor the efficiency of the development process. At the level of Linux, ASEBA seamlessly interacts with other programs thanks to its integration with D-Bus, the standard messaging middleware on modern Linux distributions.

For these reasons, we think that the integrative approach of ASEBA might be a way to overcome the stall in robotics software standardization.

References

[1] H. Bruyninckx, "Robotics Software: The Future Should Be Open," *IEEE Robotics & Automation Magazine*, vol. 15, pp. 9–11, 2008.

[2] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for Robotics: A Survey," in *2008 IEEE Conference on Robotics, Automation and Mechatronics*, pp. 736–742, IEEE Press, 2008.

[3] W. Smart, "Is a Common Middleware for Robotics Possible?," in *proc. IROS 2007 workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware*, IEEE Press, 2007.

[4] S. Magnenat, V. Longchamp, and F. Mondada, "Aseba, an event-based middleware for distributed robot control," in *Workshops DVD of International Conference on Intelligent Robots and Systems (IROS)*, 2007.

[5] S. Magnenat, P. Retornaz, M. Bonani, V. Longchamp, and F. Mondada, "ASEBA: a modular architecture for event-based control of complex robots," 2009. submitted for publication.

[6] R. Burton, "Connect desktop apps using d-bus." http://www.ibm.com/developerworks/linux/library/l-dbus.html, 2004.

[7] F. Mondada, G. C. Pettinaro, A. Guignard, I. Kwee, D. Floreano, J.-L. Deneubourg, S. Nolfi, L. Gambardella, and M. Dorigo, "SWARM-BOT: a New Distributed Robotic Concept," *Autonomous Robots, special Issue on Swarm Robotics*, vol. 17, no. 2-3, pp. 193–221, 2004.

[8] I. Standards, *Road Vehicles Interchange of Digital Information - Controller Area Network - ISO 11898*. International Organization for Standardization, 1993.

[9] M. Mizukawa, H. Matsuka, T. Koyama, T. Inukai, A. Nodad, H. Tezuka, Y. Noguch, and N. Otera, "Orin: Open robot interface for the network," in *SICE*, pp. 925–928, IEEE Press, 2002.

[10] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar, "Miro - middleware for mobile robot applications," *Robotics and Automation, IEEE Transactions on*, vol. 18, pp. 493–497, Aug 2002.

[11] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "Rt-middleware: distributed component middleware for rt (robot technology)," in *International Conference on Intelligent Robots and Systems (IROS)*, pp. 3933–3938, IEEE Press, 2005.

[12] H. Bruyninckx, "Open robot control software: the orocos project," in *International Conference on Robotics and Automation (ICRA)*, pp. 2523–2528, IEEE Press, 2001.

[13] L. Petersson, D. Austin, and H. Christensen, "DCA: A Distributed Control Architecture for Robotics," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pp. 2361–2368, IEEE Press, 2001.

[14] H. Durrant-Whyte, B. Rao, and H. Hu, "Toward a fully decentralized architecture for multi-sensor data fusion," in *Robotics and Automation, 1990. Proceedings., 1990 IEEE International Conference on*, pp. 1331–1336, IEEE Press, May 1990.

[15] J. Gil, A. Pont, G. Benet, F. Blanes, and M. Martínez, "A CAN Architecture for an Intelligent Mobile Robot," in *Proc. of SICICA-97*, pp. 65–70, 1997.

[16] I. Gravagne, J. Davis, J. Dacunha, and R. Marks, "Bandwidth reduction for controller area networks using adaptive sampling," vol. 5, pp. 5250–5255, April-1 May 2004.

[17] M. Szymanski and H. Worn, "Jamos - a mdl2ϵ based operating system for swarm micro robotics," in *Swarm Intelligence Symposium, IEEE*, pp. 324–331, IEEE Press, 2007.

[18] J. Luckham, D.C.; Vera, "An event-based architecture definition language," *Software Engineering, IEEE Transactions on*, vol. 21, pp. 717–734, 1995.

```
#!/usr/bin/python

import dbus
import dbus.mainloop.glib
import glib
import gobject
import pygame

def dbusReply():
  pass

def dbusError(e):
  print 'dbus␣error:'
  print str(e)
  loop.quit()

def scanJoystick():
  global ox, oy
  pygame.event.pump()
  x = joystick.get_axis(0) * 60
  y = -joystick.get_axis(1) * 60
  if x != ox or y != oy:
    asebaNetwork.SendEventName('SetSpeed',
      [y+x, y-x],
      reply_handler=dbusReply,
      error_handler=dbusError)
    ox = x
    oy = y
  # reschedule scan of joystick
  glib.timeout_add(20, scanJoystick)

if __name__ == '__main__':
  # inits main loop and joystick
  dbus.mainloop.glib.DBusGMainLoop(
    set_as_default=True)
  pygame.init()
  joystick = pygame.joystick.Joystick(0)
  joystick.init()
  ox = 0
  oy = 0

  # gets stub of ASEBA asebaNetwork
  bus = dbus.SessionBus()
  asebaNetworkObject = bus.get_object(
    'ch.epfl.mobots.Aseba', '/')
  asebaNetwork = dbus.Interface(asebaNetworkObject,
    dbus_interface='ch.epfl.mobots.AsebaNetwork')

  # schedules scan of joystick
  glib.timeout_add(20, scanJoystick)

  # starts event loop
  loop = gobject.MainLoop()
  loop.run()
```

Listing 4: A Python program to send speed commands. This program uses the `python-gobject`, `python-dbus`, and `python-pygame` libraries.