ViPER: a Lightweight Approach to the Simulation of Distributed and Embedded Software

Jean-Luc Béchennec
IRCCyN
1, rue de la Noë
BP 92 101, 44321 Nantes
Cedex 03, France
Jean-Luc.Bechennec@irccyn.ec-nantes.fr

Mikaël Briday
IRCCyN
1, rue de la Noë
BP 92 101, 44321 Nantes
Cedex 03, France
Mikael.Briday@irccyn.ec-nantes.fr

Sébastien Faucou
IRCCyN
1, rue de la Noë
BP 92 101, 44321 Nantes
Cedex 03, France
faucou@univ-nantes.fr

Florent Pavin
IRCCyN
1, rue de la Noë
BP 92 101, 44321 Nantes
Cedex 03, France
florent.pavin@irccyn.ec-nantes.fr

Fabien Juif
IRCCyN
1, rue de la Noë
BP 92 101, 44321 Nantes
Cedex 03, France
fabien.juif@etu.univ-nantes.fr

ABSTRACT

This paper describes a simulation platform for embedded software named ViPER (Virtual Platform and Environment Runtime). ViPER is oriented toward (but not limited to) systems of the automotive domain. It allows to model and simulate distributed embedded hardware platforms in order to ease the early development stages of the embedded software. Each node of the system is virtualized in a process that runs an ad-hoc port of the real-time operating system Trampoline. ViPER manages global time, hardware interrupt and offers a quick and easy way to model hardware devices. In order to close the loop, relevant parts of the environment can be simulated. Once a platform is modeled, ViPER generates description files for each node that ensure the conformance of the hardware abstraction layer to the virtual hardware. ViPER and Trampoline are available as free software.

Categories and Subject Descriptors

I.6.7 [**Computing Methodologies**]: Simulation and Modeling—*Simulation Support Systems, Environments*
; D.4.8 [**Software**]: Operating Systems—*Performance, Simulation*

General Terms

Design, Verification

Keywords

simulation, virtualization, embedded software

SIMUTools 2010 March 15–19, Torremolinos, Malaga, Spain.

1. INTRODUCTION

An embedded system is usually composed of four main parts: embedded software (or firmware), hardware, plant and networks. The embedded software is executed on the networked hardware platform to control the plant.

During the system specification and design, the system is considered as a whole. Then, each part is developed separately for a while. When a runnable system can be assembled, integration tests can be run. Most of the time, these tests reveal the presence of design and implementation errors and trigger a "test-and-fix" cycle. In order to limit the duration of this cycle, integration tests shall be run as soon as possible. A widely used approach to meet this requirement consists in building preliminary versions of the system composed of real and simulated parts. Of course, this makes sense only if the simulated parts are quick and easy to develop.

In this paper, we introduce ViPER, a lightweight Python framework dedicated to the simulation of distributed embedded platforms. ViPER offers a quick and easy way to develop models of hardware devices, networks and relevant parts of the plant in order to build early integration testbeds for embedded software. To be integrated in the testbed, each node is virtualized in a process of the guest operating system. The HAL[1] of the embedded software of the node communicates with ViPER through a dedicated low-level API based on POSIX IPC.

To demonstrate the viability of the approach, a ViPER port of the Trampoline[2] RTOS[3] has been developed. This port offers a solution for the rapid prototyping of Trampoline applications on workstations, as well as a solution to perform some tests that would be too difficult or too costly to realize on a real implementation.

The paper is organized as follow: in section 2, the context is presented and related works are discussed; in section 3, ViPER is presented; in section 4, the port of Trampoline to ViPER is described; in section 5, two case-studies are shown; in section 6, some con-

[1]HAL: Hardware Abstraction Layer
[2]Trampoline may be downloaded at
`http://trampoline.rts-software.org/`
[3]RTOS: Real Time Operating System

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8712
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8712

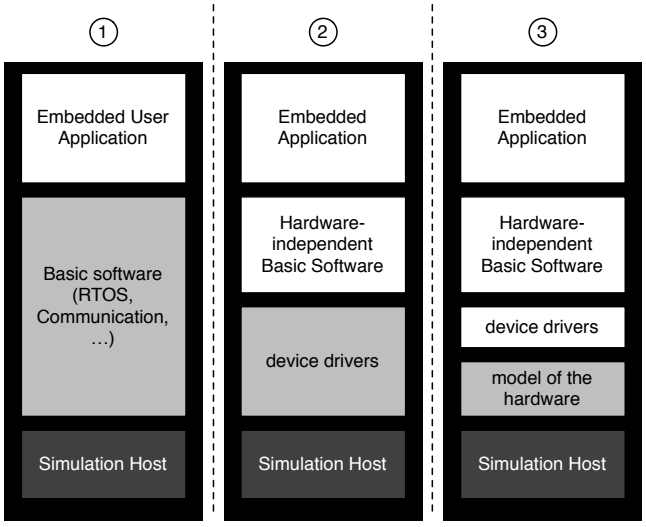

Figure 1: The three branches of SWiL approach. White rectangles represent the actual software of the simulated embedded system while light grey rectangles represent the simulated software/hardware.

cluding remarks are given.

2. CONTEXT

2.1 Using Simulation for Firmware Development

Simulation is a faithful ally for embedded system designers. Coding and running abstract models of the components of the system allow to start the validation and verification activities very early in the design process. Moreover, it allows to control the development by refining progressively these models from the specification level down to the implementation level. As a consequence, the duration of the system design process can be dramatically reduced [5].

This paper deals with the use of simulation to support the development of distributed embedded software [3]. The basic idea is to simulate the execution of the real firmwares on a model of the platform (composed of hardware, network and plant components). This kind of approach is known as "software-in-the-loop" (SWiL).

SWiL brings several benefits. Integration tests can be run before the availability of the platform. Even if the simulator is not perfectly accurate, these tests allow to detect and remove a significant number of errors in a fast and comfortable development environment. Moreover, it is generally possible to automate the execution of regression test suites using one of the scripting engines of the development platform. Another interesting point of SWiL is the possibility to run complex scenarios, easily, safely and repeatedly. For instance, it is possible to simulate precisely a specific failure mode of a device without damaging the real device.

The SWiL approach can be split into three branches, as shown in figure 1, depending on the part of real software used in the simulator.

In the first branch (① in figure 1), only the application software is used. The model of the platform is a library that maps the API of the target RTOS and communication protocols onto the API of the host (either a general purpose OS or a simulation framework). Given the distance between the semantics of both API, some features might be difficult to simulate with a satisfying degree of accuracy: scheduling (of tasks on CPU, of frames on communication channels), inputs/outputs handling (especially hardware interrupt requests), memory protection, memory mapping, and real-time. Unfortunately, these features are crucial for the design of embedded systems.

In the second branch (② in figure 1), all the hardware independent parts of the firmwares are used. The model of the platform simulates the plant, the networks and the set of devices connected to each micro-controller. For each node, an executable file is build conform to the format of the host OS. The simulated devices are accessible through a dedicated API. Thus, each set of devices is targeted as a normal development board and dedicated BSPs[4] have to be developed, including a port of the target RTOS. Solutions of this branch can be classified as host-compiled paravirtualization. Compared to the first branch, it is possible to achieve accurate simulation of features such as scheduling and inputs/outputs handling. Some features are still hard to take into account, like memory protection, memory mapping, and real-time. Tools like VxSim (Wind River) or OSE Soft Kernel (ENEA) are representative of this branch.

In the third branch (③ in figure 1), all the parts of the firmwares are used. Moreover, they are used in the form of binary images compatible with the targets (whereas the two other branches use object codes or binary codes compiled for the host). The model of the platform simulates the full system, including memories, internal buses and microprocessors. Each simulated device is accessible through an API conform to the real programming interfaces of the devices. The microprocessor models are capable of simulating the execution of binary code. Solutions of this branch can be classified as full system virtualization. With these solutions, it is possible to simulate accurately nearly all the features of the platform, except the real-time. The major difficulties encountered when using this approach is the development time of new models. Tools like SimOS [10] (Stanford University) or Simics [8] (VirtuTech) are representative of this branch.

2.2 Requirements for ViPER

Creating and using a SWiL environment for a project has a cost. This cost must be reasonable compared to the benefits brought by the environment. Therefore, the environment shall be comfortable to use, easy to setup and to control. Of course, it must be able to simulate the behavior of the real system with a good accuracy. In the context of firmware development, RTOS-level API simulation has to be eliminated for its weak accuracy. The choice has to be made between host-compiled paravirtualization and full system virtualization. The former is easier to setup and to use, while the latter is more accurate.

The work described in this paper has been conducted in the context of the development of Trampoline, an open-source RTOS inspired by the standards of the automotive domain (see section 4). The main motivation for the usage of a SWiL environment is to improve the comfort of the developers of Trampoline communication stack for compiling and testing their code. Therefore, host-compiled paravirtualization has been preferred to full system virtualization. Moreover, it has been decided that the SWiL environment has to meet the following requirements:

- It must support the modeling of a broad range of hardware devices, as well as the physical processes controlled or monitored by these devices
- It must support the design of distributed platforms by inter-

[4]Board Support Package

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8712
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8712

connecting models of network controller and models of communication media (e.g. buses or wireless channels).

- It must be able to synchronize the executions of the firmwares and the virtual platform of a distributed system.
- It must be portable on different general purpose OSes and different development environments.
- The development of a virtual platform must be easy. Models of devices must be reusable and extensible.

No existing open-source project fulfilling these requirements has been identified. Therefore, a dedicated solution has been developed. It is named ViPER (Virtual Processor and Environment Runtime). It is presented in the next section.

3. VIPER

3.1 Software Architecture

A distributed embedded systems is composed of firmwares, nodes, devices and networks. Each firmware reads (resp. writes) informations from (resp. to) the networks and devices connected to its host node. The architecture of a ViPER testbed reproduces this organization:

- each node is mapped onto a process of the host OS. Each process runs a program composed of the firmware of the node built with a dedicated BSP.
- another process runs ViPER. ViPER is in charge of executing the models of the virtual platform and synchronizing the concurrent execution of the virtual platform and the firmwares by providing a global virtual time (see section 3.2 below).
- Event channels are used between each firmware process and the ViPER process. These channels are implemented on top of POSIX IPCs[5].

ViPER is written in Python[6] and C. The core of the simulation engine is written in Python. The event channel library is written in C. It has been interfaced with Python thanks to SWIG[7] in order to connect models of the devices to event channels

The user uses Python to code the models of the components of the virtual platform and to create testbeds by interconnecting instances of models of components. It is expected that the firmwares are written in C (however, other language could be used as long as they can be interfaced with C).

This architecture is sketched on figure 2.

3.2 Providing a Global Virtual Time

In the real world, the physical time flows at the same rate in all the parts of a distributed embedded system. In a simulator conform to the software architecture described above, it is not the case. All the processes of the simulator are granted an access to a computation core for a fraction of the real-time, according to the scheduling policy of the host OS. Some processes can be blocked, waiting for a synchronization with a concurrent process. The flow of the real-time (external to the simulator) cannot be used as a basis to build the flow of the time in the simulated world.

[5]IPC: Inter Process Communication

[6]Python is a high-level interpreted programming language, available on many platforms, see `http://www.pyhton.org`.

[7]`http://www.swig.org`

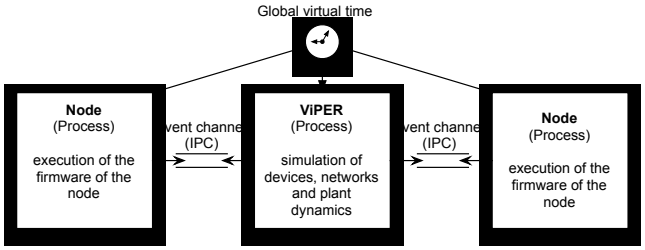

Figure 2: Software architecture for the simulation of a system with two nodes.

Thus, a virtual time has to be designed. A virtual time can be either global (the system is synchronous: the real-time is perceived identically by all the nodes) or local (the system is asynchronous: the real-time is perceived differently by the different nodes). Although real systems are asynchronous, this version of ViPER implements a global virtual time. It is easier to implement and it is an acceptable abstraction for the early design stages of distributed embedded systems. The implementation of a local virtual time service might be explored for future versions.

The simulation engine of ViPER is a discrete event simulator based on a calendar (see section 3.3.3). Building a virtual time consists in dating the virtual firing dates of the observable events when they are inserted in the calendar.

First, let us consider the events produced by the models of the components of the virtual platform. These models are designed to evolve in the virtual time. The virtual firing date of an event produced by one of these models is computed by adding the actual virtual date and the firing latency of the event (computed at runtime by the model).

Now, let us consider the events produced by the firmwares. First, even if the simulated firmwares and the real firmwares are built from the same source code (excepted the HAL), the development chains and the processor architectures are different. Therefore, the (real) execution time of a block of code in the simulator is not representative of the execution time of the same block (i.e. a block generated from the same source) in the real system. Thus, it would make no sense to compute the virtual firing date of a firmware event by adding the real response time[8] of the task that produces this event, to the activation date of this task. Second, the execution time of the concurrent tasks of the real firmware on the real platform are not known with precision at early design stages. For many systems, it is even difficult, if not impossible, to provide an accurate estimation of these execution times. Thus, the approach exposed above for platform events can not be used.

In this version of ViPER, when a firmware event is fired, it is timestamped with the current value of the global virtual clock. The timestamp must be written by the producer (firmware process) in order to cancel the unpredictable latency caused by IPCs and thread scheduling between the production and the consumption of the event. As a consequence, several firmware events can be timestamped with the same value without being fired simultaneously.

On the implementation side, the ViPER process embeds a timer thread that is used to increment the global virtual clock. The tick of this timer shall be set at a sufficiently large value to minimize the effect of the timesharing scheduler of the host OS. For instance, if the time quantum used by the host OS scheduler is q, and the

[8]Response time: latency between the activation of the task and its termination, taking into account interference of other tasks, user interrupt routines and kernel activites.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8712
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8712

typical number of concurrent threads in the system is n, the timer tick shall be at least set to $n \times q$.

3.3 Communication Between a Firmware and ViPER

In ViPER, a device driver communicates and the virtual device communicate through the event channel. Thus, the event channel must support the following communication schemes:

- *hardware interrupts* raised by devices;
- *update of status or control registers*. Notice that the behavior is not the same when the firmware writes into a control register (the device is immediately aware of this update) and when a device writes into a status register (the firmware will read the register on demand).

In order to be portable, the event channel is implemented using POSIX IPCs, especially signals, shared memory segment and semaphores. This is sketched on figure 3 and explained in the following paragraphs.

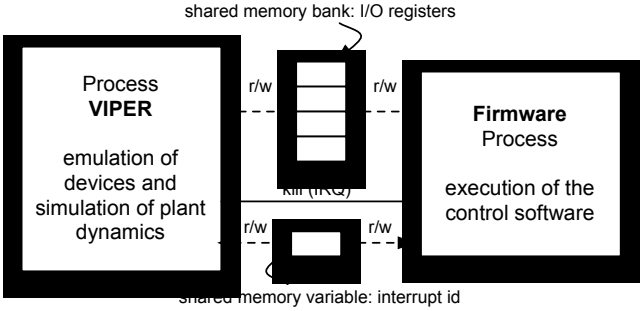

Figure 3: Details of the event channel.

3.3.1 Modeling Interrupts

To raise a hardware interrupt on a node, the event channel sends a POSIX signal to the corresponding process. Only 32 POSIX signals are available, and most of them are reserved for the operations of the host OS. To bypass this limitation, the event channel also writes the identifier of the interrupt channel in a shared variable. The identifier is used as a bit field and allows up to 64 interrupt sources, thus allowing different interrupts to occur at the same date.

Three signals may be used, in function of the priority of the interrupt source, thus simplifying the firmware implementation: `SIGALARM` for non maskable interrupts (NMI), `SIGUSR1` and `SIGUSR2` for maskable interrupts handled by the system.

The function of the event channel library used to raise an interrupt executes the following sequence of actions: it takes a mutex, updates one bit of the shared variable used to store the pending interrupts, sends the selected signal, and lastly releases the mutex.

Upon reception of the signal, the firmware execution is interrupted and the signal handler is executed. It has to call a function of the event channel library used to react to an interrupt. This function takes the mutex, makes a local copy of the pending interrupt mask, resets the mask and lastly releases the mutex.

With this protocol, an interrupt request can be lost only if a device raises two requests on the same interrupt channel before that the target firmware takes into account the first one, in the same way than in the real hardware. It can happen in the following situations:

- the model of the device evolves without letting the time flow. This model is wrong and must be corrected.
- the virtual global time is too fast compared to the real-time. The virtual global time must slow down.
- the firmware is still handling previous interrupts. This may be a firmware design error, that must be corrected.

It is possible to configure ViPER to log lost interrupt requests.

The API provided to the firmware HAL designer is quite simple. To enable or disable interrupt categories, the designer can use the standard POSIX functions (sigblock, sigmask, sigprocmask, etc.) to block and unblock signals. To obtain the current value of the pending interrupt mask, it has to call the following function:

```
unsigned int vp_ipc_get_interruption_id(ipc_t *x)
```

where `x` points to a global data structure of type `ipc_t` that contains the information necessary to access the event channel (shared memory segment and semaphores).

3.3.2 Modeling Registers

Access to devices in micro-controller is performed using specific function registers. A register is a small chunk of memory that is used to interact between the software and the hardware. There are two types of registers:

- *control registers* that are written by the application. They modify the behavior of the underlying hardware;
- *status register* are modified by the hardware and are read by the application.

ViPER offers the ability to model different devices (timers, motor through a PWM, network adapter, ...), each of them containing specific registers. This approach coincides with reality, thus facilitating the implementation by developpers.

As registers should be shared by ViPER and the firmware, they are set in an array in the shared memory. Each register has a unique index in this array. This index, that should be the same for the 2 processes, is defined at compile time (see section 3.5). The index is based on the device identifier and the register identifier, thus allowing to use multiple instances of one device.

Modeling a status register is quite simple; The firmware is not notified when ViPER writes to status register. However the access is protected by a semaphore to prevent the reading of data that are not yet completely written by ViPER. In the implementation, this is simply done in the firmware's HAL using the method:

```
reg_t vp_ipc_read_reg(ipc_t *, reg_id_t r);
```

where `r` contains both device and register identifiers, `reg_t` is the type of a register (32 bits unsigned integer).

Modeling control registers is more complicated, as ViPER has to react when the firmware writes to the register. In addition to the array of registers, a FIFO is implemented in the shared memory to notify ViPER that a register was updated. Each time that the firmware writes to a control registers, it adds an entry in the FIFO, with identifiers both the register and the device used. A reading thread in ViPER reads the new value of the register and give the information to the related device, using the producer/consumer model. It then remove the FIFO entry, position an event in the logical scheduler for the related device and waits for other notifications.

In the implementation of the firmware's HAL, the following method are provided, in the same approach than the reading method:

```
void vp_ipc_write_reg(ipc_t *, reg_id_t, reg_t);
```

Another method is used to explicitly signal to ViPER that registers are updated:

```
void vp_ipc_signal_update(ipc_t *,dev_id_t,mask_t m);
```

where `m` is a mask that indicates which registers are updated.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8712
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8712

When configuring a timer for instance, the appropriate registers are written (prescaler, reload value, ...), and then the firmware indicates to ViPER to take them into account. This is done to reduce ViPER's computation overhead.

3.3.3 Logical Scheduler

The logical scheduler is the centerpiece of ViPER because it functionally organizes the time flow: it permits to activate devices at the required moment. Each entry in the logical scheduler is called an event, and is associated to a device.

In that way, a timer will position an event in the logical scheduler at the time of the next interruption to wake up and effectively sends the interruption, a continuous model (eg an electrical engine) will position an event at each sampling periods to update the internal model. When the firmware writes to a control register, the ViPER's reading thread inserts an entry in the logical scheduler at the current date.

Therefore, when using a single processor configuration, only 2 threads are used. A reading thread that is blocked on the FIFO most of the time, and a thread that handles the scheduling of events and wakes up devices at the desired date.

3.4 Extending ViPER to Model Distributed Systems

ViPER extension for distributed systems is natural with the approach that was used for the model with a single firmware. Indeed, the presence of the logical scheduler can order events from all network nodes on a single logical time base. This ensures a *global virtual time* base for all nodes.

As in figure 2, each firmware (a network node) has its own shared memory and keep the same interface with ViPER (registers and interruptions) as in the mono-processor approach. Consequently, for a network of n nodes, there are n reading threads in ViPER and 1 thread for the logical scheduler, providing the *global time functionality*.

ViPER is adapted to provide network devices to allow the communication between different nodes. At this time, basic CAN network devices are implemented.

In the case of a reactive system with a low processor load (periodic behavior driven by the interrupts), the timings offered by the simulation are close to the timings on the real target, because of the global time approach that ensure the timing synchronization between different nodes.

On a system with a high processor load, differences between the real target and the host processor may lead to different timing behaviors because the tasks of the simulated application may miss their deadlines and the real-time behavior would be compromised. However, Viper allows to change the timebase of the logical scheduler. This slows down the pace of event processing and lowers the host processor load.

3.5 Build Process

ViPER generates 2 files from the hardware configuration for each network node:

- `vp_ipc_devices.h` is an include file that gives the order of registers that are stored in the shared memory. The order should be the same in both the firmware and ViPER. This unique file ensures this aim.

- `target.cfg` is a file that gives symbols related to interrupt handler. This file is not mandatory. When using ViPER with the Trampoline RTOS, this file is used to give symbolic names to interrupt sources to keep the application description as readable as possible.

Figure 4 highlights the build process of the whole application.

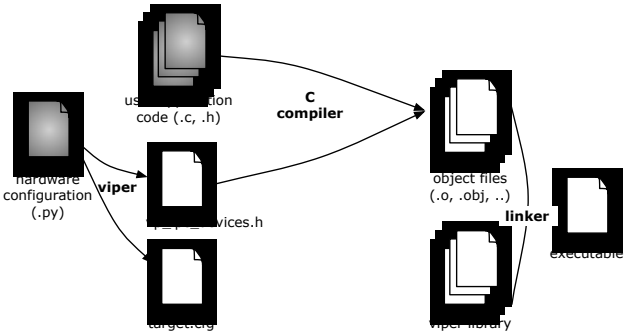

Figure 4: Build process of a Trampoline application using ViPER. ViPER generates from the hardware configuration the ordered list of registers as used in the shared memory, and gives symbol names of interrupts for an easier description of the application.

3.6 Modeling Hardware Components

ViPER aims at modeling the underlying hardware, in order to ease first development steps of an embedded software. We have seen in section 3.3 how is performed the interface between the firmware and ViPER and we focus here on how to model the hardware part.

Hardware components, as in the real world, are organized as *devices*. They have to:

- interact with the firmware through registers (section 3.3.2). Each device has a set of registers;

- react to events from the logical scheduler (section 3.3.3);

- insert events in the logical scheduler, in order to wake itself up or wake another device up in the future;

- send an interrupt to the firmware;

- allow the generation of files, as described in section 3.5.

3.6.1 Implementation

Figure 5 shows the global architecture of ViPER. The Scheduler manages an event list with dated events. The configuration gives a set of network nodes (*Ecu* stands for *Electronic Control Unit*), all of which containing one or more devices. The *Ether*s are objects which represent a system-wide infrastructure like a network or a simulated environment. An *EtherDevice* interacts with an *Ether* object. For instance, a network device sends/gets messages to/from an *Ether* object or a sensor may acquire the temperature from a simulated power-plant.

Each device derives from the `Device` superclass. Modeling a new device simply consists in writing a derived class, that implements:

- the constructor, in which registers used in the device should be given.

- the `start` method, which is called at the beginning of the simulation. It can be used to add events in the logical scheduler at startup.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8712
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8712

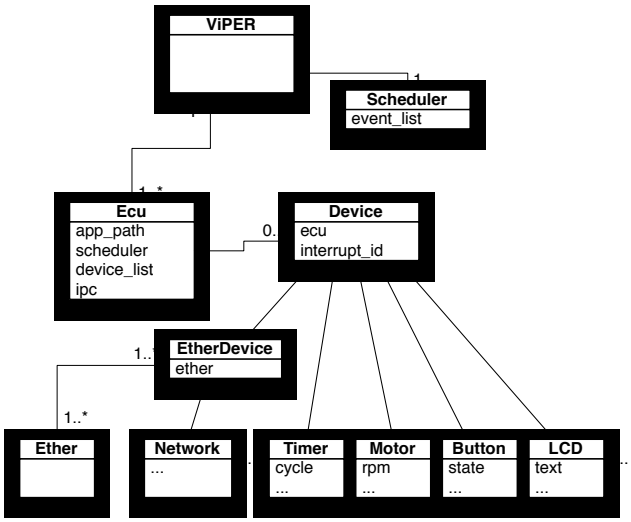

Figure 5: Simplified UML diagram of the architecture of ViPER.

- the `event` method, which is called by the logical scheduler when an event related to this devices occurs

3.6.2 Modeling a Basic Timer

To give an example, the model of a basic timer is given with the following characteristics:

- a configuration register is used `TIMCON`, where the most significant bit is a flag to run (1) or stop (0) the timer, and the last 31 bits give the periodicity of the timer.
- the timer should send an interrupt periodically, in function of periodicity given in the control register.

Here is the constructor of the basic timer device:`name` is the name of the device (`timer1`, `timer2`, ..), `id` is its internal number, used by the logical scheduler, and `signal` is the type of POSIX signal used for interruptions (see section 3.3.1):

```
class basicTimer(device.Device):
  def __init__(self,name,id,
               signal = device.SIGUSR2):
    controlReg = Register(name + "_TIMCON")
    self.__control  = controlReg.name
    self.__delay = 0
    device.Device.__init__(self, name, id,
                  signal, [controlReg])
```

The constructor instantiates a register and calls the superclass. This way, the superclass will generate the appropriate build files (see section 3.5).

Then, the firmware will be able to write to the control register of the basic timer. It will generate an event, handled by the scheduler which call the `event` method of the timer. If there is no register update when calling this method, this is because the delay of the timer elapsed:

```
def event(self, modifiedRegisters = None):
  if not modifiedRegisters: #timer running.
    self._scheduler.addEvent(
         Event(self, self.__delay))
    self.sendIt()
  else: #firmware update
    val = self._registers[self.__control].
          read()
    if val & (1 << 31) then: #run?
      self.__delay = val & ((1 << 31)-1)
      self._scheduler.addEvent(
           Event(self, self.__delay))
```

The `start` method does not need to be derived here. This example is a basic implementation and does not remove unwanted events when the firmware updates the periodicity of the timer.

4. THE TRAMPOLINE RTOS

Trampoline [1] is an open source implementation of the OSEK/VDX[9] ISO standard [6] and is evolving to encompass the future AUTOSAR standard [11].

OSEK/VDX is an industry standard for RTOS used in distributed control units in vehicles. Various parts are proposed for the standard: OS, the basic services of the real-time kernel, COM, the communication services, NM, the Network Management services and OIL, OSEK Implementation Language that is used to describe the structure of the application.

An OSEK/VDX operating system manages many objects: tasks, resources, events, alarms, counter and messages. A fixed priority scheduling policy is used with FIFO as a secondary criterion to arbitrate between tasks with the same priority. A static priority is assigned to each task when the application is built. The scheduler elects the highest priority ready task that runs until it blocks or terminates or is preempted by a higher priority task.

Tasks are execution threads and may be of two kinds: basic and extended. A basic task is activated, runs and terminates without being able to do a blocking system call. An extended task may block to wait for an event.

Resources are used to implement mutual exclusion using the OSEK-PCP protocol (Priority Ceiling Protocol). OSEK-PCP protocol is more simple than original PCP [9] [7], it is also known as *Highest Locker* protocol. When a task gets a resource, its priority is immediately raised to the resource priority (which is the maximum priority of the tasks that share the resource). So other tasks that share the same resource cannot get the CPU.

Events allow tasks to synchronize. A task may wait for a set of event and blocks if it has not been set until another task sets it. Events are private. There is no time-out built in the event mechanism but alarms (see below) may be used for this purpose.

Alarms and Counters are designed to implement recurring processing like periodic tasks. Any interrupt source may be used to increment a counter: timer, signal from mechanical organs of a car engine (camshaft, crankshaft). When a predefined counter value is reached, the associated alarm expires and the corresponding action is done (task activation, event setting or execution of a routine). An alarm may be one-shot or cyclic.

Communication services of OSEK/VDX are built around the *message* object. Two types of messages are offered: those using the blackboard model (Unqueued Message, single place buffer); those using a FIFO (Queued Message). The communication services are the same whatever the communication is local or distant.

OSEK/VDX is a statically configured operating system: all objects are known at compile time and no object may be created during the lifetime of an application. On one hand this can be seen as a limitation but on the other hand this greatly improves the predictability of a real-time application and helps to work out its memory footprint too.

[9]OSEK/VDX: Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug / Vehicle Distributed eXecutive

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8712
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8712

The development of Trampoline has started in 2005 and targets many hardware platforms like Freescale/IBM PowerPC, ARM, Freescale S12, Atmel AVR, NEC v850e and Infineon C166 now. From the start, a port targeting a Posix operating system and the support provided by ViPER has been considered to ease the development. The Posix port works using a portable multithreading engine [4].

5. CASE STUDIES

In order to demonstrate the simplicity of the configuration and the usefulness of ViPER, 2 case studies are presented hereafter. The first one is a mono-ECU[10] application that introduces how a simulated ECU and its devices are instantiated. The second one is a distributed system where each ECU controls a robot. Both applications are included in the Trampoline distribution which can be downloaded at `http://trampoline.rts-software.org`.

5.1 Analog Signal Generator

The goal of this application is to generate a rectified waveform signal on an oscilloscope while displaying the elapsed time from application startup on a LCD. This application is used in real-time labs on the actual target (Infineon C167 microcontroller) in University of Nantes.

2 tasks (processes), *Time* and *Wave*, and 2 alarms, *AlarmTime* and *AlarmWave*, are used. *Time* is activated every second by *AlarmTime*. It updates the elapsed time and display it on the LCD. *Wave* is activated every 5 ms by *AlarmWave* and sends the Y coordinates of the Waveform signal (the curve is broken in 10 segments) to a Digital Analog Converter (DAC). In addition, 2 push buttons are used to change the period of the signal. Each button sends an interrupt which triggers an ISR[11]. Each ISR decreases or increases the period of *AlarmWave* and display the new period on the LCD. Figure 6 shows the design of the whole application using the MCSE formalism [2].

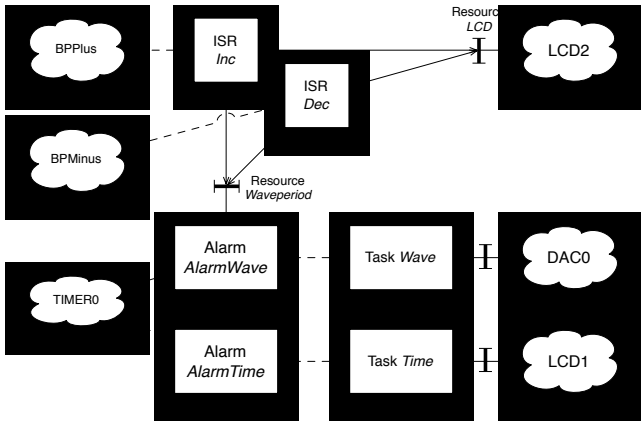

Figure 6: Design of the *Analog signal generator* application. Each cloud shape represents an environment or hardware entity. Other shapes are software entities. Arcs with an 'I' shape represents the update of a variable. When more that one producer exists for a variable, a resource is added to insure mutual exclusion. dashed arcs are events: hardware interrupts, task activation, ...

[10]Electronic Control Unit. A set of electronic devices and their associated software fulfilling a function in the automotive industry jargon

[11]Interrupt Service Routine

The following hardware devices have to be modeled: 2 push buttons, a DAC and a LCD. In addition, a timer (TIMER0) is needed to send interrupts to the Alarms. In the actual application, the DAC output is sent to an oscilloscope. In the simulated one, it is drawn on the screen in a oscilloscope-like view. In the same way, the buttons are simulated by button widgets. The following listing shows how the hardware models are instantiated and figure 7 shows the resulting graphical user interface.

```
allEcus = [
  Ecu(
    "../App-GBF/trampoline",
    scheduler,
    [
      Timer("TIMER0",
            1,
            type = timer.AUTO,
            delay = 0.01),
      DAC("DAC0", 2),
      LCD("LCD1", 3),
      LCD("LCD2", 4),
      BP("BPPlus", 10),
      BP("BPMinus", 11),
    ]
  )
]
```

The first parameter of the ECU is the binary file of the application, the second is the scheduler of Viper and the third one is a list of devices. These devices correspond to the cloud shapes of figure 6. Each device has a least 2 parameters: its name and the interrupt id it uses. Some may have more parameters like the timer: its type (one shot or automatic) and its period.

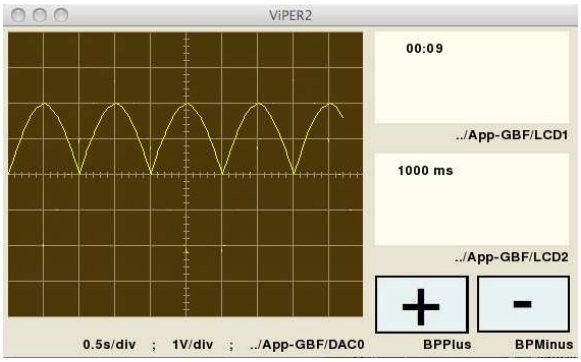

Figure 7: Viper hardware components simulation interface. Components that interact with the user may have a graphical interface. Here, the DAC value is displayed on an oscilloscope. Both LCDs are displayed too and buttons may be clicked.

5.2 Wireless Robots Ballet

The second application is a wireless robots ballet. Each robot is moved by two motors driving two wheels, one on the left and one on the right. A robot turns by driving the motors at different speed. One of the robots is the ballet master. It is controlled by the operator using a four buttons pad to accelerate, slow down and turn. Each time interval (each 100 ms for example), the ballet master reads its rotary encoders and determines its position (knowing its starting position) by odometry. For each trajectory modification by the user, the robot sends its movement vector (for now, directly

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8712
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8712

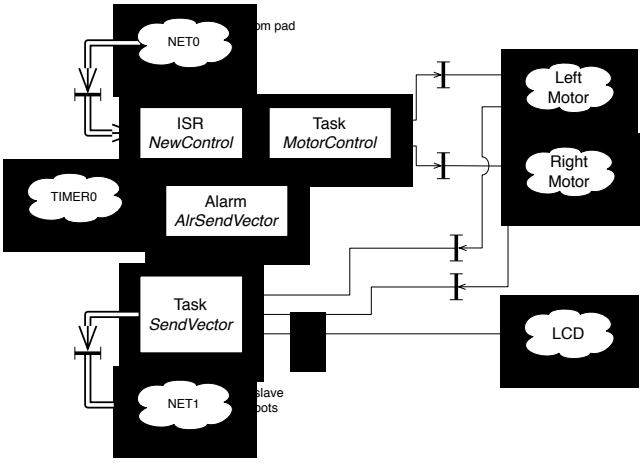

Figure 8: Design of the *Ballet Master* application. Double line arrows are messages (variables with an associated event) in the MCSE formalism.

motors commands) to the other robots wirelessly. The other robots follow the movements of the ballet master.

The *Ballet Master* application uses 2 tasks and 1 ISRs. ISR *NewControl* is triggered by the incoming command message from the pad. It activates Task *MotorControl* that changes the speed of the motors according to the command. Task *SendVector* is activated by the mean of the alarm *AlrSendVector* triggered by a timer every 100ms. It reads the positions of the motors, sends the motor positions to other robots using the wireless network and display the position on the LCD. Figure 8 shows the design of the *Ballet Master* application.

The *Slave* application have a similar design, the difference being that task *Display* reads the positions of the motors and displays them only. Figure 9 shows the design of the *Slave* application.

A least 3 ECU have to be modeled: The Pad, the Ballet Master robot and 1 of the Slaves robots. ViPER configuration is an ECU list shown below and the resulting graphical user interface is shown at figure 10.

```
allEcus = [
  Ecu(
    "../App-Pad/trampoline",
    scheduler,
    [
      Network(network_master, "NET0", 0),
      Timer("TIMER0",
            1,
            type = timer.AUTO,
            delay = 1),
      BP("BPFaster", 5),
      BP("BPSlower", 6),
      BP("BPLeft", 7),
      BP("BPRight", 8),
    ]
  ),
  Ecu(
    "../App-Master/trampoline",
    scheduler,
    [
      Network(network_master, "NET0", 0),
      Timer("TIMER0",
            1,
            type = timer.AUTO,
            delay = 0.1),
      LCDServer("LCD1", 2, display_server),
      Motor("MOTOR1_1",3),
      Motor("MOTOR1_2",4),
      Network(network_slave, "NET1", 5),
    ]
  ),
  Ecu(
    "../App-Slave/trampoline",
    scheduler,
    [
      Network(network_slave, "NET1", 0),
      Timer("TIMER0",
            1,
            type = timer.AUTO,
            delay = 0.1),
      LCDServer("LCD2", 2, display_server),
      Motor("MOTOR2_1",3),
      Motor("MOTOR2_2",4),
    ]
    ...
  )
]
```

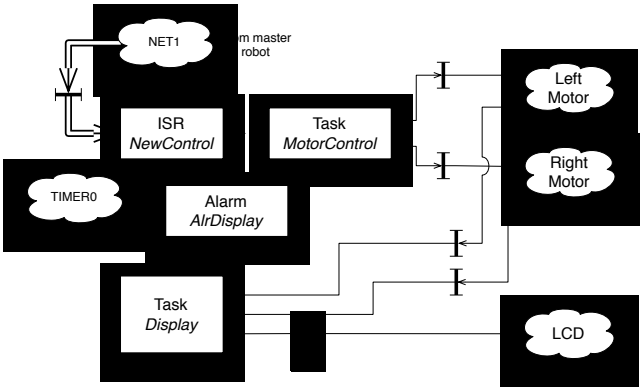

Figure 9: Design of the *Slave* application.

The first ECU is the *Pad* with 6 hardware devices: the communication network interface used to send commands to the ballet master, a timer and 6 buttons to accelerate, to slow down and to turn left and right. The second ECU is the *Ballet Master* with 6 devices: the network interface used to get commands from the *Pad*, a Timer, a LCD, 2 motors and a network interface used to send the position of the master to the slaves. The last ECU is one of the *Slaves* with 5 devices: the network interface to get the position of the master, a Timer, a LCD and 2 motors. Parameters *network_master* and *network_slave* are objects which implement the physical layer of the network.

6. CONCLUSION

This paper has presented ViPER, a SWiL environment allowing to simulate a distributed embedded application on a POSIX operating system. ViPER offers an easy to use object oriented API to model devices and to manage a global virtual time allowing to synchronize observable events among the system.

ViPER may be used to develop a firmware either from scratch or based on the dedicated port of the real-time operating system Tram-

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8712
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8712

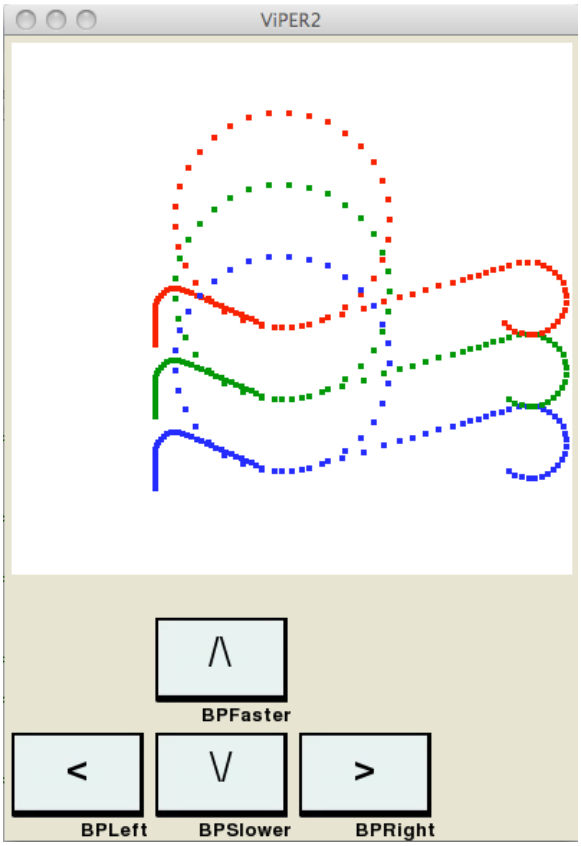

Figure 10: Wireless robots ballet components simulation interface. Here, the only hardware components displayed are the buttons of the *Pad*. A graphical view displays the positions of the robots and their trajectory in the world. The informations displayed in this view are got from the motors of all the robots. The topmost robot is the ballet master, the other two are slave robots. This kind of view is easy to implement in simulation but a headache on a real hardware.

poline to combine host-compiled para-virtualization with hardware devices models and plant models. The resulting simulation support system offers a good trade-off between the accuracy of the simulation and the development cost of hardware models and their related drivers. However, on a heavily loaded system, the user may have to slow down the simulation to keep the results accurate.

Beside the development of a library of ready to use devices and tools to observe and trace the behavior of the node, future work includes a support for fault injection in the nodes to automate tests of robustness.

7. REFERENCES

[1] J.-L. Béchennec, M. Briday, S. Faucou, and Y. Trinquet. Trampoline: An Open Source Implementation of the OSEK/VDX RTOS Specification. In *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 62–69. IEEE Industrial Electronics Society, September 2006.

[2] J. P. Calvez and D. Isidoro. A codesign experience with the mcse methodology. In *CODES '94: Proceedings of the 3rd international workshop on Hardware/software co-design*, pages 140–147, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[3] J. Engblom. Using simulation tools for embedded software development. In *Embedded System Conference*, San Jose, CA, USA, April 2008.

[4] R. S. Engelschall. Portable multithreading - the signal stack trick for user-space thread creation. In *In Proc. USENIX Tech. Conf*, pages 239–250, 2000.

[5] P. Giusto, A. Ferrari, L. Lavagno, J.-Y. Brunel, E. Fourgeau, and A. Sangiovanni-Vincentelli. Automotive Virtual Integration Platforms: Why's, What's and How's. In *IEEE Internation Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 370–378, Freiburg, Germany, September 2002. IEEE Computer Society.

[6] O.-V. Group. Road vehicles - open interface for embedded automotive applications. ISO 17356, 2005.

[7] J. W. S. Liu. *Real-Time Systems*. Prentice Hall Inc, 2000.

[8] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.

[9] R. Rajkumar and J. Lehocsky. Priority Inheritance Protocols: an Approach to Real-Time Synchronisation. *IEEE Transaction on Computer*, 39(9):1175–1185, 1990.

[10] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 7:78–103, 1997.

[11] T. Scharnhorst and al. Autosar - challenges and achievements 2005. *VDI Berichte*, (1907), 2005.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8712
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8712