PUC

ISSN 0103-9741

Monografias em Ciência da Computação

nº 02/2021

A Virtual Machine for Reactive Programming on IoT devices

Adriano Branco

Noemi Rodriguez

Silvana Rossetto

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 02/2021 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena July, 2021

A Virtual Machine for Reactive Programming on IoT devices

Adriano Branco Noemi Rodriguez
Silvana Rossetto[1]

[1]Instituto de Computação Universidade Federal do Rio de Janeiro - UFRJ

abranco@inf.puc-rio.br , noemi@inf.puc-rio.br , silvana@dcc.ufrj.br

Abstract. A large range of Internet of Things (IoT) applications use small embedded devices, combining a resource-constrained microcontroller (MCU) with a radio for wireless communication and, possibly, some sensor and actuators. Computational and memory limitations restrict the approach of creating different layers of abstractions used for conventional operating systems and libraries. The event-driven execution nature of these systems provides opportunities to save battery power, but at the cost of increasing programming complexity. A reactive programming language facilitates the development of event-driven systems in which tasks are associated with incoming events. The Terra system combines the use of ready-made, safe, components with a reactive scripting language, Céu-T. In this paper we present details of the Terra virtual machine, discussing its design for resource constrained devices. We describe how the virtual machine supports the synchronous reactive programming model of Céu-T, which triggers the execution of pending *trails* in response to external events, and how it implements the integration of the scripting language with specialized sets of components.

Keywords: Virtual Machine, IoT - Internet of Things , Reactive Programming, Embedded System, WSN - Wireless Sensor Network

Resumo. Uma grande variedade de aplicações de Internet das Coisas (IoT) usa pequenos dispositivos embarcados, combinando um microcontrolador de recursos limitados (MCU) com um rádio para comunicação sem fio e, possivelmente, alguns sensores e atuadores. Limitações computacionais e de memória restringem a abordagem de criação de diferentes camadas de abstrações usadas em sistemas operacionais convencionais e em bibliotecas. A natureza de execução orientada a eventos desses sistemas oferece oportunidades para economizar energia da bateria, mas ao custo de aumentar a complexidade da programação. Uma linguagem de programação reativa facilita o desenvolvimento de sistemas orientados a eventos nos quais as tarefas são associadas aos eventos de entrada. O sistema Terra combina o uso de componentes prontos e seguros com uma linguagem de script reativa, Céu-T. Nesta monografia, apresentamos detalhes da máquina virtual Terra, discutindo seu projeto para dispositivos com recursos limitados. Descrevemos como a máquina virtual suporta o modelo de programação reativa síncrona de Céu-T, que dispara a execução de *trilhas* pendentes em resposta a eventos externos, e como ele implementa a integração da linguagem de script com conjuntos especializados de componentes.

Palavras-chave: Máquina Virtual, IoT - Internet das Coisas , Programação Reativa, Sistemas Embarcados, RSSF - Redes de Sensores Sem Fio

In charge of publications :

PUC-Rio Departamento de Informática - Publicações
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: publicar@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Contents

1 Introduction

In the Internet of Things (IoT), *things* are mostly real-world objects connected to the Internet through embedded devices. As it expands to ever more objects, the IoT relies on networks of cheap devices which may be able to acquire information and act on the environment using sensors and actuators. These devices must provide some wireless communication mechanism, but typically have limited resources, because their cost must be sufficiently low to allow them to be deployed at large scale.

These small devices combine a resource-constrained microcontroller (MCU) with a small radio for data communication, besides other items dependent on the specific application. Because of these limited resources, it is common for devices in the IoT not to connect directly to the Internet, which would require them to run a full TCP stack. Instead, often these devices create *ad hoc* networks using small radios, and rely on a gateway to transfer information to and from the conventional Internet. To be used outside urban areas, in mobile applications, or places with wired power restrictions, they are powered by batteries or other alternative energy sources. These adhoc networks are often known as *wireless sensor and actuator networks* (WSANs). Figure 1 shows two common network typologies applied on IoT - Star and Tree/*ad hoc*.

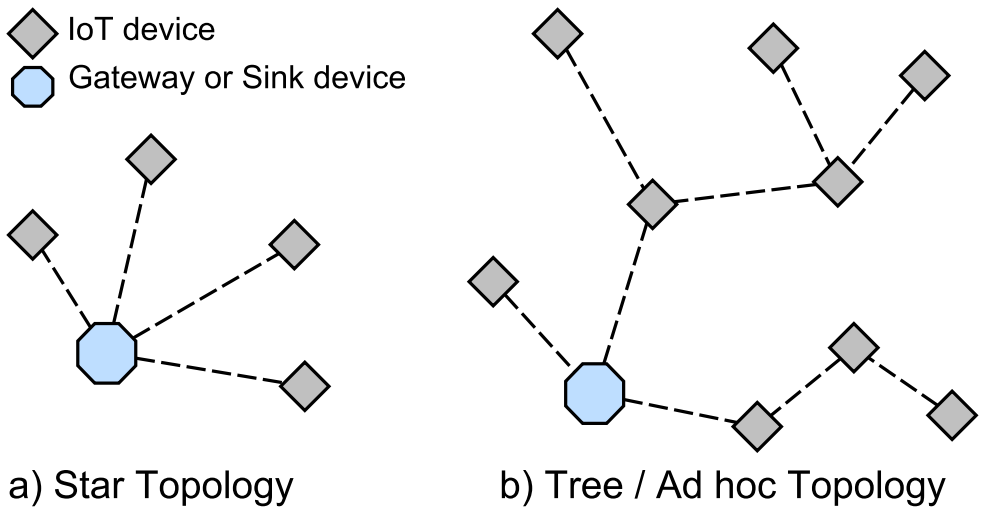

Figure 1: Common IoT network typologies

Development of applications that run in this environment poses a series of challenges. Because resources are very limited, the conventional layered programming model is inadequate, and applications must handle the specific routing mechanisms they require. Routing and other coordination tasks may involve large numbers of node and must deal with frequent rates of failure. Besides, the pace of the application is dictated mainly by external events to which it must always be ready to react [1].

In previous work, we proposed Terra [2], a system which facilitates the development of applications in IoT adhoc networks. Terra explores the fact that each specific network runs a single application to provide specialized virtual machines with ready-made components for common coordination and communication patterns. Using these components, the programmer can write applications using a reactive scripting language with several safety guarantees. This approach also facilitates application update, because the resulting scripts can be transmitted over radio with low overhead.

In this paper, we discuss the internals of the Terra Virtual Machine VM-T. In previous work [2], we reported experimental results showing how Terra simplifies programming, the small bytecode size it allows for scripts, and its energy efficiency. Here we discuss how we overcame some resource limitations and the main ideas we followed in the development of VM-T.

In the next section we introduce the main challenges related to programming small IoT devices and describe our overall approach. Section 3 presents the main characteristics of Terra. In Section 4 we present the Céu-T scripting language and its requirements for the VM-T design. Section 5 presents the VM-T architecture, focusing on the instruction set, reactive engine and the VM execution mode. Also we explain the parts of the Terra compiler that are relevant to VM-T. Next, in Section 6, we introduce the support for integration with specialized components and present a set of components ready for use in IoT applications. The current Terra implementations are presented in section 7. The evaluation of VM-T regarding processing overhead and energy consumption is reported in Section 8. Finally, Section 9 presents our final considerations.

2 Challenges and General Approach

In this section, we present the main challenges related to the programming very small IoT devices and introduce the approach we take to address part of these challenges. We also present some related work on virtual machines for IoT applications.

2.1 Main challenges for programming small devices

There are a number of programming challenges posed by networks of small wireless devices, both for communication protocols and for application programming. The scarcity of resources, along with the event-driven nature of applications and the need for coordination among large numbers of nodes makes programming applications a difficult and error prone task [3–5].

Computational and memory limitations restrict the possibility of creating different layers of abstractions that hide the work details of a subsystem. Therefore, typical solutions mix application-level problems with the lower layer levels, such as radio messaging and interaction with hardware devices. This context requires a savvy developer at both the application level and the embedded system level. Probably also an expert in distributed systems to deal with network communication protocols.

Battery operated devices have an operational lifetime based on battery charge. Reducing energy consumption will extend device uptime. The typical event-driven execution model of these systems provides opportunities to save battery power, but at the cost of increasing programming complexity [5]. In an **event-driven programming model**, the program running within each node on the network reacts to local events to perform tasks. For example, events such as a received radio message or a sensor data ready to be read must be handled by the system. This allows us to set the device to sleep mode to save power while waiting for a new event to process. This energy-saving behavior is very important when energy consumption is the critical point in devices that use battery.

Different technologies have been applied to IoT as applications evolve, resulting in a mix of different standards and creating a variety of programming environments and communication protocols. This encompasses traditional applications, such as industrial and home automation and also includes new concepts such as smart cities, smart buildings,

wearable devices and autonomous vehicles. So, the hardware of IoT devices tends to use **different types and sizes of platforms** according to the specific application. For example, small MCUs and a simple radio can be used on devices that detect simple physical values without any complex calculations, for example, temperature sensors. On the other hand, more robust MCUs and a high data rate radio can be used on devices that process images or data streaming. This may require to deal with different radio technologies and communication protocols into the same application.

Remote device configuration is also a challenge in this type of system, in special for tree/*ad hoc* networks topologies. After application deployment it may not be feasible to recover the device to reconfigure it. Remote configuration through dissemination of messages over the network allows us to change the configuration of the device when necessary. However, this brings new challenges related to network protocols, the scope of configuration and additional energy consumption.

2.2 Virtual machine approach and related works

The use of virtual machines support very well the separation of responsibilities between high-level and low-level modules while enabling cross-platform interoperability. Giving the same abstraction for different types of platforms helps to homogenize the user view of the system. Furthermore, it is also possible to extend the virtual machine with a set of specialized components to support low-level services, such as common communication patterns. Thus, developers may benefit from a high-level programming environment given by the virtual machine and its specialized components. At same time, the virtual machine engine may accommodate the task execution model to better support the desired event-driven model.

Different works propose frameworks or architectures to support the programming of IoT and WSAN applications by using virtual machines to address the problem of programming layer abstraction [3, 6–12].

To our knowledge, the first work proposing the use of virtual machines in wireless sensor networks is Maté [6]. The Maté VM is built on TinyOS [13] and has a very simple instruction set. The code propagation and execution is broken up into 24 instructions called capsules. A capsule fits into a single message packet. Maté limits its context execution to only three concurrent paths, one for sending messages, another one for receiving messages, and a third one for a timer. Maté has up to 8 user-defined instructions that enable additional virtual machine customization and its operand stack has a maximum depth of 16. To address some of Maté's limitations the Maté team built ASVM [7]. ASVM is an application-specific virtual machine. The authors proposed a custom runtime machine to support different application-specific high-level languages, but each language needs its own compiler. ASVM implements a central concurrency manager to support the sequential execution on concurrent handlers. This is an optional service to help user applications avoid race conditions. This solution assumes that handlers are short-running routines that do not hold on to resources for very long.

DAViM [8] is very similar to ASVM but adds the possibility of parallel execution. DVM [9] is based on the application-specific VM concept from ASVM, but it uses SOS [14] as its operating system. SOS allows dynamic loading of system modules. In DVM, it is possible to load different combinations of high-level scripting languages and low-level runtime modules. DVM [9] and DAViM [8] also use a concurrency manager like ASVM's. Inspired on TinyDB [15], SwissQM [16] has a query-specific instruction set and a high-level language similar to SQL.

Cosmos and Regiment implement customizable VMs with high-level languages that are specifically designed for WSNs. Cosmos [3] uses mPL as high-level language and mOS as operating system. The scripting language is limited to the data flow control using the custom mOS functions. Cosmos also allows dynamic loading of new C functions. In Cosmos, an event handler uses only local variables and its data are exchanged by input/output interface queues. These characteristics avoid race conditions. Regiment [12,17] uses a reactive functional language with a special semantic for intra-network operations. The runtime implements the basic operations and access to devices. In Regiment, an event handler task run to completion and cannot be blocked. This also avoids race conditions.

Other examples of virtual machine-based solutions for addressing small devices programming include PICOBIT [18] and Pascal P5 VM [19]. More recently, several alternatives have emerged to facilitate the programming of larger capacity devices like the Espressif family ESP8266 [1] and ESP32. Some examples are WArduino [20] with a port of WebAssembly, NodeMCU [21] running e-Lua [22], MicroPython [23] with a subset of Python language, and Espruino [24] running JavaScript.

2.3 General approach

In this paper, we present a new virtual machine-based approach, called Terra, designed to resource constrained devices. Terra combines the use of specialized components with a reactive scripting language. The components expose a desirable abstraction level for the programmer while the reactive scripting language allows code to be statically analyzed in order to avoid unbounded execution and memory conflicts, contributing to safer code. A reactive programming language is, unlike an imperative programming language, a natural approach to programming an event-driven system where reactions (tasks) are associated with incoming events [25].

In the next two sections, we first present an overview of the Terra system, followed by an explanation of the Terra script language.

3 Terra Overview

Terra [2] provides a component-based virtual machine VM-T to be specialized for different application domains. Applications are written in Céu-T, a variant of the Céu programming language [26]. Céu-T defines a specific execution model which VM-T needs to support. Because our initial focus was on WSN devices, we first implemented Terra on the MicaZ [27] and TelosB [28], but VM-T has proved to easily portable to other architectures. The current implementation runs on a few different platforms. The footprint of a VM-T with a minimum of components uses about 40k bytes of ROM and requires only 4k Bytes of RAM to run small scripts. VM-T memory requirements for different implementation are explained in subsection 7.1.

Figure 2 presents the three basic elements of Terra. The following sections present an example of a script written in Céu-T and an overview of the VM-T architecture. Component specializations are discussed later in Section 6.

[1] ESP8266 specs- 80 MHz Xtensa LX3 32 bit CPU, TCP/IP stack, GPIO pins and 512 KiB to 4 MiB flash memory

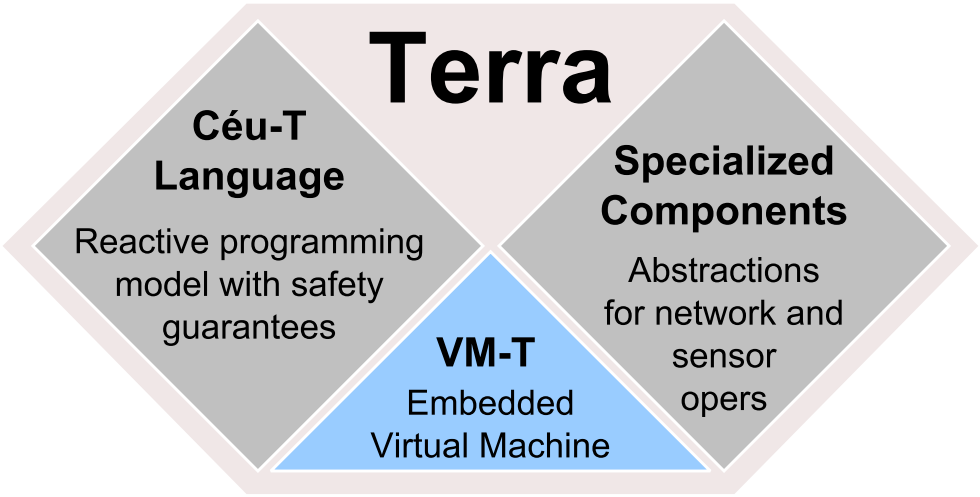

Figure 2: Terra system basic elements.

3.1 Script example

The example in Figure 3 shows a script that invokes complex operations embedded in the VM-T runtime. This application supposes there is a set of nodes forming a monitoring network: Each node in the network topology monitors the local temperature and, if the reading is above a certain threshold - 55^{o}C, asks for its one-hop neighbors to send their current readings - `AGGREG()`, and sends the average of the collected results to the sink node (basestation) - `SEND_BS()`.

Besides declarations not represented in this example, the code contains a main loop in which it awaits for 10 seconds so as to maintain periodic readings. Each component invocation is split in an emit/await pair. The example shows that the abstractions provided by VM components for communication and routing allow the script to be concise. These components are presented in subsection 6.3. This specific example, after compilation, uses 105 bytes for bytecode and 112 bytes for data and flow control.

```
1  loop do
2      await 10s;
3      emit REQ_TEMP();
4      var ushort tValue = await TEMP();
5      if (tValue > 55) then
6          emit AGGREG(agA);
7          var aggDone_t data;
8          data = await AGGREG_DONE;
9          dataMsg.average = data.value;
10         emit SEND_BS(dataMsg);
11     end
12 end
```

Figure 3: Céu-T code for group average alarm

3.2 VM-T Architecture

The Terra virtual machine (VM-T) is composed by three modules as shown in Figure 4. The *VM* module is the main module. It provides an interface for receiving new application code from the *Basic Services* module and three interfaces for specialized events and functions. The *Engine* submodule controls the execution of code interpreted by the *Decoder* submodule and handles external events received from the *Event Queue* submodule.

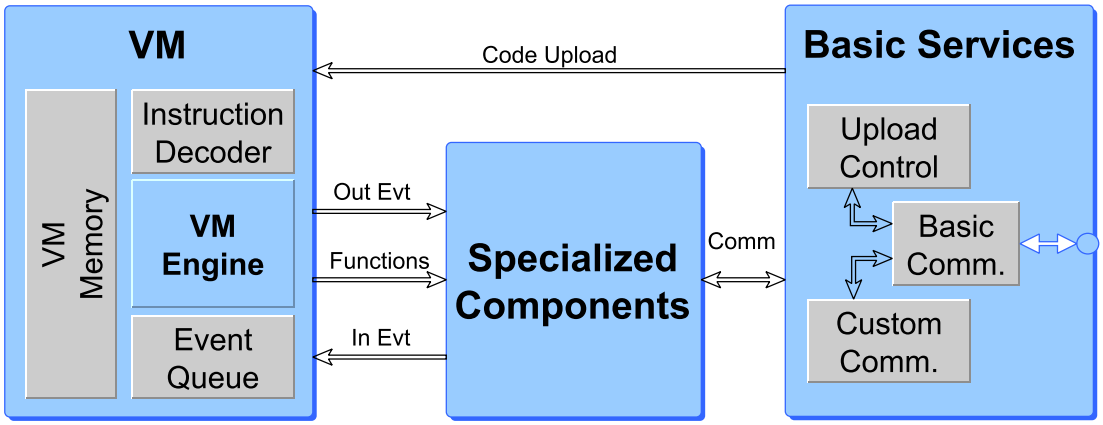

Figure 4: VM-T modules

The *Basic Services* module contains a minimum communication module and a code dissemination protocol. The code dissemination protocol makes it is possible to load new Terra scripts remotely. The communication primitives can either be used directly by the script or as building blocks for specialized components. The *Upload Control* submodule controls the dissemination protocol and loads code into VM program memory.

The *Specialized Components* module implements specific specializations of Terra. The developer of new specialization must only implement custom events and functions inside this module and write a corresponding configuration file to be used by a Céu-T script. Also, a basic specialization of VM-T already available can be used as a starting point to include new events and functions.

4 The Céu-T Scripting language

Céu-T provides parallel statements to control program flow, and has special commands to handle interaction with the environment through input and output events. The type system and expressions in Céu-T are designed for the demands small embedded devices applications and are in accordance to Terra's safety requirements. In the next subsections we describe the main Céu-T characteristics and the requirements they generate for the implementation of VM-T. For a complete description of the language, please see the full paper on Terra [2]

4.1 Céu-T events and system calls

In Terra, the Céu-T language acts as a glue between components written in *C* and embedded in the VM-T. Céu-T and components in VM-T communicate through *system calls*, *output events* and *input events*. System calls and output events cross the script boundary

towards the VM components, while input events go in the opposite direction, crossing the VM boundary towards the script. The *emit* and *await* commands are used, respectively, to signal output events and to wait for incoming events. Systems calls are part of component interfaces and are handled as functions. Any Céu-T expression can contain a system call. The *await* command blocks the execution of the current code segment until either that event occurs or the code segment is canceled.

In Céu-T, system calls are the only way to escape compiler verification. Since the system calls available to the programmer are defined and limited to those in component interfaces, it is feasible to ensure that these run in bounded time (e.g., do not contain recursive calls and infinite loops). Timers are a special case of input events defined by an amount of time unit. For example, the `await 10s` command blocks the execution of the current code segment by 10 seconds.

Céu-T has a special syntax for a configuration file in which all available events and system calls are defined for a specific specialization. The developer of a new VM must provide the VM binary and the corresponding configuration file to be included by the user. This allows the creation of new VM-T specialization without the need for a new Terra compiler.

4.2 Céu-T flow-control structures

Programs in Céu-T are designed by composing blocks of code through sequences, conditionals, loops, and parallel blocks. The combination of parallelism with standard control flow enables hierarchical compositions, in which self-contained blocks of code can be deployed independently. To illustrate the expressiveness of compositions in Céu-T, consider the two variations of the structure in Figure 5.

```
loop do                          loop do
  par/and do                       par/or do
    <...>                            <...>
  with                             with
    await 1s;                        await 1s;
  end                              end
end                              end
```

Figure 5: Compositions in Céu-T.

In the `par/and` loop variation, the code block in the first trail (represented as $<...>$) is repeated at intervals of at least 1 second, because both trails must terminate for the enclosing `par/and` to terminate and allow the loop to restart. In the `par/or` loop variation, the termination of any of the trails will cause the `par/or` to terminate and the loop to restart. In this case, code $<...>$) will be restarted at intervals of at most one second. These structures represent, respectively, sampling and timeout patterns, which are typically found in IoT applications.

Scripts in Céu-T follow the synchronous concurrency model, that is, reactions to input events run to completion and never overlap: in order to proceed to the next event, the current event must be completely handled by the script. To ensure that scripts are always reactive to incoming events, the synchronous model relies on the guarantee that a reaction always executes in bounded time. The Céu-T compiler statically verifies that programs contain only bounded loops (i.e., loops that contain an `await` statement in ev-

ery possible execution path). Also, although Céu-T supports multiple lines of execution, accesses to shared memory are safe. Because programs can react to only one component-triggered event at a time, the Céu-T compiler performs a flow analysis to detect concurrent accesses: if two accesses to a variable can occur in reactions to the same event and are in parallel trails, then the compiler issues an error message.

4.3 Céu-T Requirements for VM-T

A Céu-T script is composed by *controls structures* and *trails*. Each external event triggers the execution of all trails that are waiting for that specific event. Each trail executes until it reaches an `await` command. At that time, the runtime must execute the next pending trail, if it exists. A trail is thus the minimal execution unit of Céu-T.

The Terra compiler decomposes the user script applications translating each trail into a sequence of instructions terminated with the `end` instruction. VM-T then runs each of them, as separate execution units, to completion. The engine controls the events and the program flow using special control registers defined during compile time. All data, expressions and controls registers are statically defined during Céu-T compilation phase. This allows an early configuration of the full memory allocation and the stack usage. In the current implementation of VM-T, a trail is executed as a task of a custom scheduler. When an event arrives in the input event queue, the event control triggers the execution of each pending event trail. Basically, the engine first looks for pending trails, then looks for any pending timers and after that looks for any pending received event. If there is no pending task, the system will enter sleeping mode until it receives a new trigger or timer event.

Céu-T timers are controlled by the `await tValue` command. The wall-clock model defined by Céu strives to trigger each timer as near to the real requested time as possible. The VM-T implementation is based on an internal timer controlled by an internal hardware clock. This internal timer is set to trigger the next active timer. If any delay occurs, this is compensated in the next setting of the timer. Setting the internal timer to the next active timer, instead of having a constant cycled timer, also enables the system to go to sleep mode when idle.

5 VM-T Instruction Set Architecture

The VM-T is a virtual machine that supports bytecode execution, like the Smalltalk or the Java virtual machine. Although the VM-T Instruction Set Architecture (ISA) is very simple, it can be classified as a complex instruction set computer (CISC). All instructions are implemented in *C* and compiled to the target platform. The VM-T engine supports different expression operations and specific Céu-T flow control instructions. The input and output operations are performed by generic instructions for function calls and events.

The engine does not have built-in generic registers, but it does have a stack to support expression operations and to pass arguments in function calls. Some operations can access memory data directly or indirectly. When required, the instruction arguments are passed in sequence. Special registers are the Program Counter (PC) and the Stack Pointer (SP).

All memory allocations, both for flow control and for program variables, are mapped at compile time. The stack maximum size is also computed at compile time. When the VM-T engine starts a new run, the full memory allocation is known beforehand. The user

will be able to validate at compile time whether the target platform will accommodate the memory size required by their script.

5.1 Instruction set

The VM-T instruction set was defined to privilege small size of generated bytecode script. Although the engine is based on stack operations, the instruction set has alternative instructions to directly access data memory and constants. Another approach to gain in code size was the use of some instruction bits to define the arguments size. As an example, when using memory address values up to 255, the argument needs only one byte instead of the default two-byte address. We have similar size optimizations for constant values that can vary from one to four bytes. These are feature used by other processors and also seen on Intel 64 and IA-32 Architectures Instruction Format [29]. We've exhausted the 8-bit range (256 instructions) and we still need, in some cases, to use an additional byte to represent the instructions options. The Table 1 presents the instruction set mnemonics.

Expressions are supported by the arithmetic and logic instructions represented in the Groups A to E, the assignments instructions in Group F and the stack instructions in Group H. The `inc` and `dec` instructions are exceptions that can not be used inside expressions. The instructions `cast` and `deref` are mostly used internally by the compiler to operate on the contents of memory variables.

All instructions that operate on arrays, like `poparr` or `setarr`, check array bounds when the array index is passed as a variable argument. An internal event error is generated and the operation is ignored if the index value points out of the array's bounds. This event can be captured by the user's application script.

The execution control group includes some basic flow-control instructions and specific instructions to support the Céu-T execution model. The `exec`, `ifelse`, and `end` instructions are used in the traditional flow-control (Group J). The Calls instructions (Group I) are used in function and event calls. The `getextdt` instruction is used to recover the data of an input event. The Track (Group K) and Timers (Group L) instructions are more specific instructions that access the control registers or internal control variables. These control registers maintain the trails entry points. When an input event or a timer trigger occurs the engine looks for the trail entry point in the control registers. Similarly, a track register maintains the entry point for a trail not triggered by an event, i.e a trail in a parallel statement. Also, instructions `set`, `memcpy`, and `memclr` are used by the compiler when it is necessary to initialize or clear some data range like control registers.

5.2 Bytecode example

We now present the pre-allocated memory structure and the segments of the bytecode generated for the script in Figure 3 of Subsection 3.1. Figure 6 shows the control registers and data memory allocation. The numbers in the beginning of each line represent the data memory address. In this example we have two reserved track structures. `track 0` is a default track register that is always created for the system's main trail. `track 1` is the track register for executing the script itself. In this example we have only one running trail at each moment in time, requiring only one additional track register. Next, three control registers were created, respectively, for the three invocations of `await`: one for the timer (wClock 0), and two for the input events TEMP and AGGREG_DONE. Next, we have the allocation of all variables created in the user script. The `$ret` variable is an

Table 1: VM-T Instruction Set

A – Math oper.	
neg	Negative
sub	Subtraction
add	Addition
mod	Modulo
mult	Multiplication
div	Division
B – Comparison oper.	
eq	Equal
neq	Not equal
gte	Greater than or equal
lte	Less than or equal
gt	Greater than
lt	Less than
C – Logical oper.	
lor	Logical or
land	Logical and
lnot	Logical not
D – Binary oper.	
bnot	Binary not
bxor	Binary xor
bor	Binary or
band	Binary and
lshft	Left shift
rshft	Rigth shift
E – Special oper.	
inc	Increment var
dec	Decrement var
cast	Type casting
deref	Pointer deref.

F – Assignment	
set	Set variable value
setarr	Set array elem. value
G – Memory range	
memcpy	Copy data structure
memclr	Clear memory data
H – Stack oper.	
pop	Pop to a variable
poparr	Pop to an array elem.
push	Push a variable
pusharr	Push an array elem.
I – Calls	
func	Call a function
outevt	Call an output event
getextdt	Copy event data
J – Flow-control	
exec	Jump the execution
ifelse	Conditional jump
end	End of trail
nop	No operation
K – Tracks	
tkins	Set a track reg.
tkclr	Clear a track reg.
asen	Set a delayed track reg.
trg	Generates an event
chkret	Check par termination
L – Timers	
clken	set a timer

internal variable created automatically.

Figure 7 shows side by side the script body and the respective assembly code. The numbers before some instructions represent the memory address of a trail entry point. Each trail runs up to the `end` instruction. In this listing it is easy to see that a Terra script is compiled to a set of trails. The execution of the script is controlled based on control registers, such as those shown above, and on special instructions included in the trails. For example, the instruction at line 2 (`clken 10000 126`) sets a timer of 10,000 milliseconds to execute the trail at the memory address 126. At the timer fired event, the engine will start the execution at address 126 (line 4). `outevt REQ_TEMP` will invoke the temperature reading function asynchronously. When the execution reaches line 5, instruction `set short 10 133` will write the address value 133 to the TEMP Control Register. As shown in Figure 6, memory address 10 is part of the TEMP Control Register. When the Temperature sensor component triggers the TEMP event, the engine looks for the TEMP Control Register and starts a new trail execution at address 133. Section 6 explains the integration between an Event Control Register and an Event Component.

```
** Track Registers
——  track register 0
——  track register 1

** Control Registers
000  wClock 0
008  inEvt TEMP
012  inEvt AGGREG_DONE
```

```
** Variables
016  $ret
017  gr1
027  agA
040  dataMsg
057  tValue
059  data
```

Figure 6: Control and Data memory allocation example

```
1  loop do
2      await 10s;
3
4      emit REQ_TEMP();
5      tValue = await TEMP();
6
7
8      if (tValue > 550) then
9
10
11
12         emit AGGREG(agA);
13
14         data = await AGGREG_DONE;
15
16
17         dataMsg.average = data.value;
18         emit SEND_BS(dataMsg);
19     end
20 end
```

```
118 clken 10000 126
    end
126 outevt REQ_TEMP
    set short 10 133 //Set TEMP reg.
    end
133 getextdt tValue len=2
    push 550
    push tValue
    gt
    ifelse 147 118
147 push &agA
    outevt AGGREG
    set short 14 156 //Set AGGREG_DONE
    end
156 getextdt data len=11
    set dataMsg.average data.value
    outevt SEND_BS dataMsg
    exec 118
```

Figure 7: Assembly example - The left is a code in Céu-T and the right is the equivalent code in assembly.

5.3 Céu-T Compiler

The implementation of the Céu-T compiler is based on the Céu [30] compiler implementation. The compiler was written em Lua programming language and uses the LPeg library [31] for pattern-matching. From this base implementation we inherit all the static checking. The compiler checks scripts for non-deterministic memory accesses and tight loops (loops without *awaits*), besides other properties, such as whether all possible block cancellations are correctly captured. The compiling process uses the C preprocessor (cpp) to allow inclusion of header files, macro expansions, conditional compilation, and line control.

The main modifications for Céu-T are the inclusion of variable types and the bytecode generation. Other modifications include the addition of arithmetic and logical expressions (Céu relies on the C compiler for expressions), and some checks and code optimizations. The absence of pointers in the Céu-T type system avoids all kind of references to external variables and also avoids memory leaking. Checking types on assignments further enhances safety.

During the compilation phase, the bytecode generator performs several checks to select the most appropriate instruction combination for a specific user code. The compiler optimizer also searches for specific instruction combinations, replacing these combina-

tions with smaller sets. Whenever possible, code generation prioritizes accesses to memory over use of the stack. Expressions with binary operations like sum or minus always need to use the stack.

Listing 1: A code optimization example.

```
1  /*** Not optimized and using stack ***/
2  push &v2          : opcode
3                    : addr2Low
4                    : addr2High
5  push &v1          : opcode
6                    : addr1Low
7                    : addr1High
8  setshort          : opcode
9  total of 7 bytes of code
10
11 /*** Optimized ***/
12 setshort &v1, &v2 : opcode
13                   : addr2Low
14                   : addr1Low
15 total of 3 bytes of code
```

Listing 1 shows an example of optimization for a simple assignment like `v1 = v2;`. Considering both as `short` type variables and with memory addresses below 256 (i.e needing only 1-byte). In this example, the first part (lines 2–8) pushes to the stack two 16 bits addresses for each variable and, the last instruction, pops these addresses to copy the contents of one address to the other address. The second part (lines 12–14) uses only one instructions that does all the work without using the stack, using 8-bit addresses. In this example, the optimization replaces seven bytes of non-optimized code with three bytes of optimized code

6 Specializing VM-T

In this section, we describe the interfaces Terra provides for creating specialized VM-Ts and explain how the integration with functions and events works. We then present the library of components that is readily available with the system.

6.1 Events Interface

The engine of VM-T deal differently with three kinds of Céu-T events: external output event, external input events, and internal events. External events are generated outside the engine module, and may be device-internal events, like a sensor ready to be read, or device-external events, like a received radio message. Internal events are script variables that follow a special semantic of Céu to generate events.

The set of possible external events must be defined during the VM-T specialization. This is, in general, directly associated to the set of elements available in the hardware, such as sensors, actuators, or radios. The Terra system also allows the VM-T developer to define external events assigned to high-level operations designed for specific specializations. For example, we can have a particular complex computation that is better per-

formed in C than in Céu-T. The Volcano application [32, 33] is a good example of usage of complex computation. The Terra report [34] evaluate the VM-T components with different level of implementations for the Volcano application.

In Céu-T we use the `emit` command to call an output event and the `await` command to register a trail handler to a specific input event. Output external events may have an argument to pass simple values or data structures. Input external events may return a simple value or a data structure. This may include an additional byte value as argument to be used as the event identifier.

All external events are internally identified by one byte. This allows us to define up to 255 output events and 255 input events. Event-slot registers are used to control the input events. Similar to other control registers, all event-slot registers are defined during compilation time (a slot is created for each `await inEvent` inside the user script). These slots store the event identifier and the respective trail entry-point to handle the event. When an event has an identifier, the slot also stores this value.

The compiler translates the `emit outEvent()` command to a specific instruction passing the event identifier and, when necessary, an additional argument. Then, when executing, the engine calls the specialized internal function to execute the respective operation. This call is synchronous and the engine waits for its return to continue to the next instruction. In the other direction, the `x = await inEvent()` command is translated to a general-use memory write instruction. It updates the event-slot register indicating that there is a trail ready to handle this event. When an event is read from the engine input queue, the event control searches for both the respective event identifier and the auxiliary identifier. The control looks for all possible matches to set the track registers with each stored entry-point address. In that way, all active `await`s for an specific event will be triggered. In the case of any returned data, a special instruction is used to copy the event data buffer to the user script variable.

Céu-T allows the creation of internal events based on variables. One can `emit` and `await` a specific event variable. These internal events are used to execute a specific trail as a synchronous call broken up in different trails. From the user's point of view, an `emit x` command will jump to the code defined at the `wait x` command, and, when this trail finishes, will return to the point exactly after the original `emit` command. A set of internal variables are automatically created to control these internal events. General-use memory write and read instructions are used to manipulate these variables and the track instructions are used to trigger the executions.

6.2 Functions Interface

Functions in Terra system are very similar to external output events. The main differences are that we do not use the `emit` command, function can have several arguments and invocations to them can be used inside expressions. All functions are specialized by the VM-T developer and have an internal predefined one-byte integer identifier that allows the system to have up to 255 different functions. At the engine level, a function is a special instruction with one byte argument. Like for the output event, the engine calls the specialized internal function to execute the respective operation. This call is again synchronous and the engine waits for it to return to continue to the next instruction. Unlike output events, all arguments and returned values are passed by the stack. This allows any function to be used inside expressions.

6.3 Specialized Components

Terra offers a basic library of components that can be included (or not) in a specific virtual machine. As far as possible, these components are parameterized for genericity. New components can also be included by programmer-savvy in order to create abstractions for new programming patterns, but the goal of this basic library is to offer a set of components that is sufficient for a range of common applications. This is feasible because most applications for sensoring and actuation are variations of a basic monitoring and control pattern. We organized the needed functionality in four areas: **communication** – support for radio communication among sensor nodes; **group management** – support for group creation and other control operations; **aggregation** – support for information collection and synthesis inside a group; **local operations** – support for accessing sensors and actuators. Currently Terra has two fully operational specialized packages – TerraGrp and TerraNet.

The TerraGrp specialization gives full support to all functionalities in the four areas: communication, group management, aggregation, and local operations. With TerraGrp, the programmer has access to high-level abstractions like message routing, group formation, leader election, and automatic value aggregation.

An alternative specialization is the TerraNet version. This specialization only gives support to basic functionalities for communication and local operations. TerraNet offers only basic communication components to send and receive messages within radio range. This allows the programmer to write Céu-T applications that uses a specific communication protocol. This is useful, for instance, to allow specific applications to decide how they will handle faults or even routing. The macro system can be used to allow other parts of the application to use the high-level protocol as if it were defined by components. This gives the programmer more flexibility to experiment with different communication and fault-handling services, which may possibly later become new components.

7 VM-T current implementations

Our first implementation of VM-T was based entirely on TinyOS. After that, it was possible to isolate the main components of VM-T from the auxiliary components of TinyOS, thanks to the NesC component model. NesC generates standard C code that can be compiled on any platform.

With the resulting architecture, a base VM-T implementation for a new platform requires a task scheduler, a timer and a communication interface. In some cases, we can use the TinyOS task scheduler itself. In general, the timer and communication implementation rely on platform-specific support. This opens up the range of Terra applications, as we can easily use any communication interface. For example, we can use any radio interface or even an UDP/IP or TCP/IP implementation. The main idea is to extend Terra utilization to others heterogeneous platforms targeting general-use IoT devices.

Terra is also appropriate for heterogeneous IoT networks, due to the possibility of running on different types/sizes of microcontrollers and using different radio technologies. Special nodes with two or more different radio technologies may act as gateway between different networks, relaying messages from on network type to another. Another approach is to use two base station nodes with different radios and interconnect them over the host serial interface. This allows messages from one radio to be redirected to the other radio. Table 2 presents the platforms on which Terra is currently available.

These implementations can be found on the Terra development website[2].

Table 2: Target platforms

Target Platform	CPU (Bits/Clk/RAM) Clk:Hz, RAM:Bytes	Radio Standard	Full TinyOS
Mica2	8 / 8M / 4K	CC1000	yes
Mica2Dot	8 / 4M / 4K	CC1000	yes
MicaZ	8 / 8M / 4K	802.15.04	yes
TelosB	16 / 8M / 10K	802.15.04	yes
IrisMote	8 / 8M / 8K	802.15.04	yes
ArduinoMega	8 / 16M / 8K	NRF24L01	no
Raspberry Pi2	32 / 700M / 512M	WIFI	no
Linux	>32 / >1G / >512M	WIFI	no
ESP8266	32 / 80M / 96K	WIFI	no
Android	>32 / >1G / >512M	WIFI	no

7.1 Terra memory usage

Using the virtual machine approach in the typical Harvard architecture of MCUs, we have to load and execute the VM-T runtime in the ROM space and allocate part of the RAM memory to load the script bytecode and the variables. Besides, the VM-T runtime needs some RAM space for its execution. As we increase the embedded specialized components, the use of ROM and RAM by VM-T is also increased. Consequently, the memory space for the Céu-T script decreases. Some hardware platforms have memory limitations that may restrict the use of specific configurations. Table 3 presents the Terra memory configuration for different small hardware platforms.

Table 3: Terra memory usage

	Mem.	MicaZ	Mica2	TelosB
TerraNet	ROM	40.0k	37.3k	35.0k
	RAM	3.6k	3.5k	7.5k
	Script space	2.0k	2.0k	6.0k
TerraGrp	ROM	55.3k	52.4k	47.1k
	RAM	3.6k	3.5k	7.8k
	Script space	0.768k	0.768k	4.8k

Units in bytes

ROM utilization depends on the CPU type and the specific component implementations for each hardware. The RAM value represents the memory used by variables in VM-T and in the system components, including the total memory allocated for the Céu-T script. This is not the full RAM size because we need to leave some room for the C stack.

When writing a Céu-T program, it is important to verify the amount of memory used. The line *Script space* in Table 3 shows how much memory is left to the application programmer for the Céu-T script in each of the specializations we explored. For example, TerraNet on MicaZ has about 2k bytes for the Céu-T script program, but TerraGrp has only 768 bytes on the same platform. This happens because the TerraGrp components

[2]Terra website - http://afbranco.github.io/Terra/terra-home.html

use more RAM than the TerraNet components, consequently leaving little memory to the user script. Because TerraGrp offers higher-level abstractions, the programmer should typically need smaller scripts in this specialization [34].

8 VM-T overhead benchmark

In this section, we describe experiments that estimated the overhead incurred by interpretation with VM-T. We compared computing-intensive code written in Terra with code written in the nesC programming language[3]. Next, we describe the test scenarios and results. At the end of Section we present an analysis of energy consumption. A full report on experiments can be found in the Terra paper [2] and in the Branco's thesis [34].

8.1 Test Scenarios — Introduction

We used three different tests to evaluate the overhead incurred by the VM as compared with direct nesC/TinyOS execution. In the first test, we ran a simple CPU-bound application: a loop that continuously increments a value. This would be an extremely uncharacteristic pattern for sensor network applications, which typically pass through relatively long intervals of quiescence, followed by short periods of activity, triggered by external events. The idea of this test was to stress the processing capacity of VM-T to the limit. In the second test, we measured the overhead of the VM bounded by an IO operation. In this case the application repeatedly reads data from a sensor in a closed loop waiting the sensor response. In the third test, we measured the VM overhead in a more typical scenario, in which the application repeatedly reads data from a sensor in a periodic timed loop.

In each test, we ran both variants of the application for five minutes. Every ten seconds, all applications send the value of the loop counter via radio to the base station.

In both systems, programs are coded with event-based loops. In Terra, because a tight loop is forbidden, we use a custom event to break the loop with an `await` command. In the corresponding Terra specialized component, the return event is generated immediately from the request. In the nesC/TinyOS version, each iteration posts a task representing the following iteration.

To compare the results, we used two metrics. The first one is the total number of iterations executed along the five minutes that the applications are left running. This number is the value of the counter sent to the base station at time 300s. The goal of using this metric — which can be measured both in real motes and in the simulator — is to have a rough idea of the relative processing speeds of the two platforms. The second metric we used was the total number of clock cycles in *Active* and *Idle* state[4]. The values for this metric were obtained through the simulations on Avrora [35] and helped us to understand the difference in the processing time.

[3] All program versions, including the Terra runtime, were compiled to MicaZ platform using the same radio transmission power (CC2420_DEF_RFPOWER=7).

[4] TinyOS keeps the CPU in idle state when the task queue is empty. The CPU goes into active state when it receives an interruption.

8.2 Test scenario 1 - CPU-bound Application

In this test, a simple CPU-bound application runs a tight loop that continuously increments a value. Table 4 presents the results obtained with Avrora simulator for our first scenario.

Table 4: CPU-bound Test

Metric	Program Version		
	Terra(a)	**nesC**(b)	b/a
loop counter	597,511	11,735,607	19.64
active cycles	2,175,061,049	2,174,060,892	1.00
idle cycles	37,735,747	4,768	0.0

As expected in loops with no blocking operations, the CPU was kept busy almost 100% of the execution time. The cost of interpretation becomes explicit in the value of the loop counter obtained after 300 seconds. The TinyOS version ran 19.64 times the iterations executed by the Terra version.

We also executed this same test directly on a MicaZ mote. The relation between the values obtained for the loop counter were quite close to the ones from the simulation. (Values were respectively 600,692 and 11,735,309.)

We next estimated the number of cycles per instruction in VM-T. The main loop of our test script translates to six instructions in the virtual machine. We divided the total number of CPU cycles by the final value of the counter (number of times that the loop was executed) to obtain the number of CPU cycles per loop iteration, and then divided this result by 6 to estimate the number of cycles per instruction. The result is 607 cycles, which is close to the value of 550 cycles reported for DVM (section 4.1 §2 of [9]) and not so far from the 400-cycles value obtained in the micro-benchmark of ASVM (section 4.5 §2 of [7]).

8.3 Test scenario 2 - IO-bound application

In this test, the application repeatedly reads the sensor and increments the loop value when the sensor returns a value. The CPU has to wait for the *done* event from the sensor before executing the next reading. Table 5 presents the results for this scenario.

Table 5: IO-bound Test

Metric	Program Version		
	Terra(a)	**nesC**(b)	b/a
loop counter	27,269	29,999	1.10
active cycles	265,848,122	104,726,668	0.39
idle cycles	1,959,251,305	2,068,673,827	1.06

In this case, predictably, CPU active time was much less than in the first test scenario. CPU was idle around 88%-95% of the time. The nesC variant executed approximately 10% more iterations than the Terra variant. As regards CPU cycles, however, the Terra version needed around 2.5 times the cycles used by nesC. In Terra, CPU was active 11.95% of the time, while in nesC only 4.82%.

Direct execution on the MicaZ mote again produced results close to the simulator's: the value of the counter was 27,270 for the Terra version and 29,999 for the nesC one.

In Terra, approximately 91 iterations were executed per second. In ASVM, in a similar test using a mica mote [36], the ratio of 312.5 iterations per second was obtained (5000 loops per 16.0 sec in section 4.5 §4 of [7]). The difference in values was apparently due to the analog-digital conversion in sensor readings, as in our case the number of iterations was the same order of magnitude of the direct execution over nesC/TinyOS.

8.4 Test scenario 3 - IO-timer application

In this test, the application repeatedly reads the sensor every 10 seconds, increments the loop value when the sensor returns a value, and sends this value via radio. Table 6 presents the results for this test scenario.

Table 6: IO-timer Test

Metric	Program Version		
	Terra(a)	**nesC**(b)	b/a
loop counter	30	30	1.00
active cycles	9,630,812	8,896,252	0.92
idle cycles	2,239,073,188	2,239,807,748	1.00

In this case, as expected, CPU active time is much less than in the first two scenarios. The CPU was idle around 99.6% of the time. The nesC variant and the Terra variant executed exactly the same number of iterations. As regards CPU cycles, however, the Terra version needed around 1.08 times the cycles used by nesC. In Terra, the CPU was active 0.43% of the time, while in nesC 0.41%.

Direct execution on the MicaZ mote again produced similar results to the simulator's: the value of the counter was 30 for the Terra version and 30 for the nesC one.

The results for this third test scenario give us an important insight about the real costs incurred by interpretation. Although the execution of interpreted code is more expensive than that of the native nesC code, this difference practically disappears in a periodic timer pattern.

8.5 Energy consumption analysis

Table 7 shows the values of energy consumption that were reported at the end of execution of all three test scenarios using the Avrora simulator. The energy values are shown in Joules and represent total consumption in Terra and in nesC. We analyse only the two major energy consumers, radio and CPU. For the radio, we separate the energy consumption in the receive and the transmit modes.

As expected, the radio energy for the receive mode is a constant value of 0.0076 Joules per CPU Cycle. This means that energy spent in the receive mode was the same in all tests and that the use of virtual machine doesn't impact this value. In general, applications must use some mechanism to reduce the energy utilization of the radio in receive mode, like the LPL-Low Power Listening [37].

All three nesC executions used 0.0014 Joules to transmit the same 30 radio messages. In Terra, considering the energy spent in the received mode and in the transmission, we

Table 7: Energy consumption results

		Total CPU Cycles	Energy (in Joules) CPU	Receive	Transmit
CPU-bound	Terra	2,212,796,796	6.75	16.86	0.0018
	nesC	2,173,038,370	6.69	16.56	0.0014
IO-bound	Terra	2,225,099,427	3.48	16.96	0.0019
	nesC	2,172,629,320	3.14	16.56	0.0014
IO-timer	Terra	2,222,949,986	3.04	16.94	0.0018
	nesC	2,185,740,922	2.99	16.66	0.0014

had a small energy overhead incurred by the code dissemination protocol, but this is negligible in a long running application.

The difference in the amount of energy consumed by the CPU is due to the difference in the periods of activity. In the documentation of the Atmel microcontroller [38], table *DC Characteristics* in pages 318/319 indicates that an active cycle consumes roughly 2.5 times the energy consumed by an idle cycle. The IO-bound test for Terra had 2.5 times the number of active cycles used by nesC. However, because the total number of active cycles still remains small in proportion to the number of idle cycles, energy consumption was only 11% higher. In the IO-timer test, Terra had only 1.09 times the number of active cycles used by nesC and the energy consumption was only 1.7% higher. This overhead would typically diminish, possibly to negligible rates, in real applications, in most of which the active/idle ratio is very small. Part of this overhead is due to the cost of code dissemination, and would also typically diminish in long running applications.

9 Final Remarks

In this work, we discussed the VM-T virtual machine which is part of the Terra system. Terra was designed to support the development of IoT applications, especially those that communicate over wireless networks (WSN). Developing IoT application involving small devices raises several challenges for system programming. Some of these challenges are the microcontroller (MCU) resource constraints, the typical event-driven programming model, the need for remote configuration via radio network, the limited power capacity provided by batteries, and the different types and sizes of platforms.

Terra uses a reactive scripting language combined with a set of specialized components that help address these key challenges. The reactive scripting language supports the event-driven programming model and favors the reduction of battery power consumption. Direct integration with specialized components helps to avoid a multi-layer system that generally doesn't fit the available resources. This also exposes the user to an abstraction layer that facilitates application programming. At the same time, these components can pre-implement complicated network protocols.

We have shown how the VM-T engine implements the reactive execution model of Terra programs, including some programming safety guarantees. Also, the generic event interface allows easy integration to new specialized components. In addition, we have reported the impact of virtual machine related to processing cost and memory usage. The low memory footprint allows the use of Terra in very small microcontrollers. We have ported Terra to platforms with very different architectures and sizes. This shows that it is possible to achieve interoperability between different platforms using compatible

communication protocols across different radio technologies. We have also measured the energy overhead impact for different processing regimes. For a typical IoT application, this impact had only a 1.7% increase in energy consumption.

Overall, the development of VM-T has shown that the approach of combining a reactive scripting language with pre-programmed components to create a safe programming environment in which the developer can create applications by gluing these components together is feasible even in devices with very limited resources. This seems to be a promising approach to deal with the ever-growing need for IoT applications.

References

[1] HAREL, D.; PNUELI, A.. **On the development of reactive systems**. In: Apt, K. R., editor, LOGICS AND MODELS OF CONCURRENT SYSTEMS, p. 477–498, 1985.

[2] BRANCO, A.; SANT'ANNA, F.; IERUSALIMSCHY, R.; RODRIGUEZ, N. ; ROSSETTO, S.. **Terra: Flexibility and safety in Wireless Sensor Networks**. ACM Transactions on Sensor Networks, 11(4):59:1–59:27, Sept. 2015.

[3] AWAN, A.; JAGANNATHAN, S. ; GRAMA, A.. **Macroprogramming heterogeneous sensor networks using Cosmos**. In: PROCEEDINGS OF THE 2ND ACM SIGOPS/EUROSYS EUROPEAN CONFERENCE ON COMPUTER SYSTEMS 2007, EuroSys '07, p. 159–172, New York, NY, USA, 2007. ACM.

[4] KOTHARI, N.; GUMMADI, R.; MILLSTEIN, T. ; GOVINDAN, R.. **Reliable and efficient programming abstractions for wireless sensor networks**. PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, p. 200–210, 2007.

[5] MOTTOLA, L.; PICCO, G. P.. **Programming wireless sensor networks: Fundamental concepts and state of the art**. ACM Computing Surveys, 43(3):19:1–19:51, Apr. 2011.

[6] LEVIS, P.; CULLER, D.. **Maté: a tiny virtual machine for sensor networks**. In: ASPLOS-X: PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, p. 85–95, New York, NY, USA, 2002. ACM.

[7] LEVIS, P.; GAY, D. ; CULLER, D.. **Active sensor networks**. In: PROCEEDINGS OF THE 2ND CONFERENCE ON SYMPOSIUM ON NETWORKED SYSTEMS DESIGN & IMPLEMENTATION - VOLUME 2, NSDI'05, p. 343–356, Berkeley, CA, USA, 2005. USENIX Association.

[8] MICHIELS, S.; HORRÉ, W.; JOOSEN, W. ; VERBAETEN, P.. **DAViM: a dynamically adaptable virtual machine for sensor networks**. In: PROCEEDINGS OF THE INTERNATIONAL WORKSHOP ON MIDDLEWARE FOR SENSOR NETWORKS, MidSens '06, p. 7–12, New York, NY, USA, 2006. ACM.

[9] BALANI, R.; HAN, C.-C.; RENGASWAMY, R. K.; TSIGKOGIANNIS, I. ; SRIVASTAVA, M.. **Multi-level software reconfiguration for sensor networks**. In: PROCEEDINGS OF THE 6TH ACM & IEEE INTERNATIONAL CONFERENCE ON EMBEDDED SOFTWARE, EMSOFT '06, p. 112–121, New York, NY, USA, 2006. ACM.

[10] KOSHY, J.; PANDEY, R.. **VMSTAR: synthesizing scalable runtime environments for sensor networks**. In: PROCEEDINGS OF THE 3RD INTERNATIONAL CONFERENCE ON EMBEDDED NETWORKED SENSOR SYSTEMS, SenSys '05, p. 243–254, New York, NY, USA, 2005. ACM.

[11] WELSH, M.; MAINLAND, G.. **Programming sensor networks using abstract regions**. In: NSDI'04: PROCEEDINGS OF THE 1ST CONFERENCE ON SYMPOSIUM ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION, p. 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[12] NEWTON, R.; MORRISETT, G. ; WELSH, M.. **The Regiment macroprogramming system**. In: IPSN '07: PROCEEDINGS OF THE 6TH INTERNATIONAL CONFERENCE ON INFORMATION PROCESSING IN SENSOR NETWORKS, p. 489–498, New York, NY, USA, 2007. ACM.

[13] LEVIS, P.; MADDEN, S.; POLASTRE, J.; SZEWCZYK, R.; WHITEHOUSE, K.; WOO, A.; GAY, D.; HILL, J.; WELSH, M.; BREWER, E. ; CULLER, D.. **TinyOS: An operating system for sensor networks**. In: AMBIENT INTELLIGENCE. Springer Verlag, 2004.

[14] HAN, C.-C.; KUMAR, R.; SHEA, R.; KOHLER, E. ; SRIVASTAVA, M.. **A dynamic operating system for sensor nodes**. In: PROCEEDINGS OF THE 3RD INTERNATIONAL CONFERENCE ON MOBILE SYSTEMS, APPLICATIONS, AND SERVICES, MobiSys '05, p. 163–176, New York, NY, USA, 2005. ACM.

[15] MADDEN, S. R.; FRANKLIN, M. J.; HELLERSTEIN, J. M. ; HONG, W.. **TinyDB: an acquisitional query processing system for sensor networks**. ACM Transactions on Database Systems, 30(1):122–173, 2005.

[16] MUELLER, R.; ALONSO, G. ; KOSSMANN, D.. **SwissQM: Next generation data processing in sensor networks**. In: THIRD BIENNIAL CONFERENCE ON INNOVATIVE DATA SYSTEMS RESEARCH, 2007.

[17] NEWTON, R.; WELSH, M.. **Region streams: functional macroprogramming for sensor networks**. In: DMSN '04: PROCEEEDINGS OF THE 1ST INTERNATIONAL WORKSHOP ON DATA MANAGEMENT FOR SENSOR NETWORKS, p. 78–87, New York, NY, USA, 2004. ACM.

[18] ST-AMOUR, V.; FEELEY, M.. **PICOBIT: A compact scheme system for microcontrollers**. In: Morazán, M.; Scholz, S.-B., editors, IMPLEMENTATION AND APPLICATION OF FUNCTIONAL LANGUAGES, volumen 6041 de **Lecture Notes in Computer Science**, p. 1–17. Springer Berlin Heidelberg, 2011.

[19] TEAM, P. P.. **Pascal p5 homepage**, 1986. [Online; accessed 1-June-2021].

[20] GURDEEP SINGH, R.; SCHOLLIERS, C.. **Warduino: A dynamic webassembly virtual machine for programming microcontrollers**. In: PROCEEDINGS OF THE 16TH ACM SIGPLAN INTERNATIONAL CONFERENCE ON MANAGED PROGRAMMING LANGUAGES AND RUNTIMES, MPLR 2019, p. 27–36, New York, NY, USA, 2019. Association for Computing Machinery.

[21] TEAM, N.. **Nodemcu homepage**, 2014. [Online; accessed 1-June-2021].

[22] BOGDAN MARINESCU, D. S.. **e-lua homepage**. [Online; accessed 1-June-2021].

[23] GEORGE, D. P.. **Micropython homepage**, 2015. [Online; accessed 1-June-2021].

[24] WILLIAMS, G.. **Espruino homepage**, 2014. [Online; accessed 1-June-2021].

[25] BENVENISTE, A.; CASPI, P.; EDWARDS, S.; HALBWACHS, N.; LE GUERNIC, P. ; DE SIMONE, R.. **The synchronous languages 12 years later**. Proceedings of the IEEE, 91(1):64 – 83, jan 2003.

[26] SANT'ANNA, F.; RODRIGUEZ, N.; IERUSALIMSCHY, R.; LANDSIEDEL, O. ; TSIGAS, P.. **Safe system-level concurrency on resource-constrained nodes**. In: PROCEEDINGS OF THE 11TH ACM CONFERENCE ON EMBEDDED NETWORKED SENSOR SYSTEMS, SenSys '13, p. 11:1–11:14, New York, NY, USA, 2013. ACM.

[27] MEMSIC. **Mib600 datasheet**. Product folder, 2009.

[28] MEMSIC. **Micaz datasheet**. http://www.memsic.com/wireless-sensor-networks/MPR2400CB, 2009. Accessed: July,2016.

[29] CORPORATION., I.. **Intel 64 and IA-32 Architectures - Software Developer's Manual, Vol. 2, Instruction Set Reference, A-Z**. Intel Corporation., 2016.

[30] SANT'ANNA, F.; IERUSALIMSCHY, R.; RODRIGUEZ, N.; ROSSETTO, S. ; BRANCO, A.. **The design and implementation of the synchronous language cÉu**. ACM Trans. Embed. Comput. Syst., 16(4), July 2017.

[31] IERUSALIMSCHY, R.. **A text pattern-matching tool based on parsing expression grammars**. Software – Practice & Experience, 39(3):221–258, Mar. 2009.

[32] TAN, R.; XING, G.; CHEN, J.; SONG, W.-Z. ; HUANG, R.. **Quality-driven volcanic earthquake detection using wireless sensor networks**. In: REAL-TIME SYSTEMS SYMPOSIUM (RTSS), 2010 IEEE 31ST, p. 271–280, Washington, DC, USA, Nov 2010. IEEE.

[33] TAN, R.; XING, G.; CHEN, J.; SONG, W.-Z. ; HUANG, R.. **Fusion-based volcanic earthquake detection and timing in wireless sensor networks**. ACM Transactions on Sensor Networks, 9(2):17:1–17:25, Apr. 2013.

[34] BRANCO, A.. **Scripting customized components for Wireless Sensor Networks**. PhD thesis, PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO - PUC-RIO, September 2015.

[35] TITZER, B. L.; LEE, D. K. ; PALSBERG, J.. **Avrora: scalable sensor network simulation with precise timing**. In: PROCEEDINGS OF THE 4TH INTERNATIONAL SYMPOSIUM ON INFORMATION PROCESSING IN SENSOR NETWORKS, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[36] HILL, J. L.; CULLER, D. E.. **Mica: A wireless platform for deeply embedded networks**. IEEE Micro, 22(6):12–24, Nov. 2002.

[37] POLASTRE, J.; HILL, J. ; CULLER, D.. **Versatile low power media access for wireless sensor networks**. In: PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE ON EMBEDDED NETWORKED SENSOR SYSTEMS, SenSys '04, p. 95–107, New York, NY, USA, 2004. ACM.

[38] ATMEL. **ATMEGA128**. Atmel, San Jose, CA, USA, 2467x–avr–06/11 edition, 2011.