PUC–Rio

Proposta de Tese de Doutorado

Safe and High-level Programming for Embedded Systems

Autor:
Francisco Sant'Anna

Orientadores:
Roberto Ierusalimschy
Noemi Rodriguez

June 29, 2012

Contents

1 Introduction

Embedded systems combine hardware, software, and possibly mechanical devices to perform a specific dedicated task. They differ from general-purpose systems, which are designed with flexibility in mind and encompass a multitude of applications in a single system. Examples of embedded systems range from simple MP3 players to complex fly-by-wire avionic systems. Usually they have low tolerance to faults, are constrained in memory and processing, and must conform with real-time requirements.

Embedded systems are essentially reactive as they interact permanently with the surrounding environment through input and output devices (e.g. buttons, timers, touch displays, etc.).

Software for embedded systems is usually developed in C, and the addition of a real-time operating system may extend it with preemptive and/or cooperative multithreading (MT). However, concurrency in C requires a low-level exercise related to the life cycle of activities (i.e. creating, starting, and destroying threads), besides extra efforts for explicit scheduling (in cooperative MT) and manual synchronization (in preemptive MT). Furthermore, these models lack safety warranties, given that cooperative MT is susceptible to unbounded execution, while preemptive MT is subject to race conditions and deadlocks.

An established alternative to C in the field of safety-critical embedded systems are the reactive synchronous languages [3]. Two major styles of synchronous languages have evolved: in the *control–imperative* style (e.g. Esterel [5]), programs are structured with control flow primitives, such as parallelism, repetition, and preemption; in the *dataflow–declarative* style (e.g.

Lustre [9]), programs can be seen as graphs of values, in which a change to a value is propagated through its dependencies without explicit programming.

We believe that embedded systems development can benefit from a new programming language that unifies both control and dataflow reactive synchronous styles, while preserving typical *C* features that programmers are familiar with, namely shared memory concurrency.

The central idea of our thesis is to pursue the *least common denominator* between the two styles, i.e., a programming language with the smallest set of primitives that supports both control and dataflow synchronous styles.

In our proposed programming language, by starting from a small synchronous control core and extending it with shared memory, we can derive dataflow programming without any specific primitives for this purpose. The synchronous core is very similar to the Esterel programming language [5]. On top of it, we introduce controlled side effects, in which variables can be shared among multiple lines of execution, and the compiler is responsible for detecting race conditions at compile time.

Besides offering high-level reactive programming, a primeval goal in our work is to ensure the correctness of programs through safety warranties. Our proposed semantics enables a compile-time analysis that detects unbounded loops and concurrent access to variables.

The static analysis precludes any dynamic support in the language, such as memory allocation, a call stack, and dynamic loading. However, this trade-off seems to be favorable in the context of embedded systems, as dynamic features are discouraged due to resource limitations and safety requirements.

We have a working implementation of our model as the *Céu programming language*, which is available online[1]. Céu is targeted at highly constrained embedded platforms, such as Arduino and wireless sensor nodes.

2 Related work

In this section, we present a review of some synchronous languages and techniques that relate to our research.

Event-driven programming

At the lowest abstract level of the synchronous model, event-driven programming is usually employed as a technique in general-purpose languages with no specific support for reactivity. Because a single line of execution and stack are available, programmers need to deal with the burden of manual stack management and inversion of control. [1]

[1] `http://www.ceu-lang.org/`

In the context of embedded systems, the programming language *nesC* [8] offers event-driven programming for the TinyOS [11] operating system. The concurrency model of *nesC* is very flexible, supporting the traditional serialization among callbacks, and also asynchronous callbacks that interrupt others. To deal with race conditions, *nesC* supports atomic sections with a similar semantics to mutual exclusion in asynchronous languages.

Cooperative multithreading

Cooperative multithreading is an alternative approach to preemptive multithreading where the programmer is responsible for scheduling activities in the program (known as *coroutines* [13] in this context). With this approach, there are no possible race conditions on global variables, as the points that transfer control in coroutines are explicit (and, supposedly, are never inside critical sections).

Protothreads [7] offer very lightweight cooperative multithreading for embedded systems. Its stackless implementation reduces memory consumption but precludes support for local variables. Furthermore, Protothreads provide no safety warranties besides being race-free: a program can loop indefinitely, and access to globals is unrestricted.

Finite state machines

The use of finite state machines (FSMs) is a classic technique to implement reactive applications, such as network protocols and graphical user interfaces. A contemporary work [12], based on the Statecharts formalism [10], provides a textual FSM language targeting Wireless Sensor Networks.

FSMs have some known limitations. For instance, writing purely sequential flow is tedious, requiring to break programs in multiple states with a single transition connecting each of them. Another inherent problem of FSMs is the state explosion phenomenon. To alleviate this problem, some designs support hierarchical FSMs running in parallel [12]. However, adopting parallelism precludes the use of shared variables.

Dataflow

Dataflow programming [9, 2, 14] differs from the traditional "Von Neumann" imperative style, where programs are defined as sequences of steps. With a declarative style, dataflow programs define high-level dependency relationships among data. The language is responsible for scheduling activities that propagate external changes into the dependency graph that represents a program.

Embedded systems are typically characterized by control-intensive applications, where programs have to deal with low-level I/O and handle explicit state. In this context, dataflow programming does not provide proper abstractions, being more suitable for data-intensive applications.

Esterel

Our proposed language is strongly influenced by the Esterel language [5], which also provides an imperative reactive style with a similar set of parallel compositions.

The following example illustrates the programming style of Esterel:

```
1: module ABRO:
2: input A, B, R;
3: output O;
4: loop
5:   [ await A || await B ];
6:   emit O
7: each R
8: end module
```

The program awaits the input signals `A` and `B` in parallel (line 5). After they both occur, regardless of the order, the program emits the output signal `O` (line 6) and restarts the process. If the input signal `R` occurs (line 7), the loop also restarts, without emitting `O`.

In Esterel, the semantics for time is similar to that of digital circuits, where an external clock defines discrete steps in which multiple signals can be active.

Esterel has no support for shared-memory concurrency, and "if a variable is written in a thread, then it can be neither read nor written in any concurrent thread". [4]

3 Proposed model

In our proposed language, multiple lines of execution—known as *trails*—continuously react to input events from the environment. Waiting for an event suspends the running trail until that event occurs. The environment broadcasts an occurring event to all active trails, which share a single global time reference (the event itself).

To illustrate the concurrent reactive nature of the language, the example in Figure 1 executes three trails in parallel through the `par` statement. The program increments a variable every 1 second, which may be reset by an external event. Every time the variable changes, its current value is displayed on screen.

Lines 1-3 declare the variables and events used in the program. The declarations include the type of value the occurring event communicates. For instance, the external event `Restart` carries integer values, while the internal event `changed` is a notify-only event, holding no values.

```
 1:  input int Restart;     // an external event
 2:  event void changed;    // an internal event
 3:  int v = 0;             // a variable
 4:  par do
 5:     loop do             // 1st trail
 6:        await 1s;
 7:        v = v + 1;
 8:        emit changed;
 9:     end
10:  with
11:     loop do             // 2nd trail
12:        v = await Restart;
13:        emit changed;
14:     end
15:  with
16:     loop do             // 3rd trail
17:        await changed;
18:        _printf("v = %d\n", v);
19:     end
20:  end
```

Figure 1: An example.

The loop in the first trail (lines 5-9) waits for 1 second, increments variable `v`, and notifies changes through the `emit` statement (line 8). The loop in the second trail (lines 11-14) resets `v` to the value of every occurrence of the input event `Restart` (line 12), and notifies these changes (line 13). The loop in the third trail (lines 16-19) shows the value of `v` (line 18) whenever the event `change` is emitted (line 17).

The execution model is grounded on a precise definition of time as a discrete sequence of external input events: a sequence because only a single input event is handled at a time; discrete because a complete reaction always executes in bounded time (to be discussed further).

3.1 Safety warranties

One of our goals is to provide safe concurrent programming. Considering the context of embedded systems, it is important that all safety warranties are given at compile time. Currently, we propose two compile time verifications that enforce safety restrictions.

The first verification ensures that loops in the language do not run in unbounded time making programs unresponsive to handle upcoming input events. We propose that each possible path in a loop body must contain at least one `await` or `break` statement. For instance, based on this restriction, the following loops are refused at compile time:

```
// ex. 1:
loop do
   v = v + 1
end
```

```
// ex. 2:
loop do
   if v then
      await A;
   end     // else does not await
end
```

Conversely, the following loops should be accepted:

```
// ex. 3:
loop do
   await A;
end
```

```
// ex. 4:
loop do
   if v then
      await A;
   else
      await B;
   end
end
```

By structural induction on the program AST, it is trivial to infer whether a given loop body satisfies that restriction or not.

The second verification ensures that access to shared-memory never occurs concurrently in different trails running during the same reaction.

For instance, in the following nondeterministic example, the assignments run concurrently:

```
int v;
par do
    v = 1;
with
    v = 2;
end
```

while in

```
input void A, B;
int v;
par do
    await A;
    v = 1;
```

```
input void A;
int v;
par do
   loop do
      await A;
      await A;
      v = 1;
   end
with
   loop do
      await A;
      await A;
      await A;
      v = 2;
   end
end
```

Figure 2: A nondeterministic program.

```
with
    await B;
    v = 2;
end
```

there is no possible concurrency between the assignments, as A and B are external events and, by definition, cannot happen at the same time.

We propose a static analysis at compile time that generates a deterministic finite automata representing a program that covers exactly all possible points it can reach during runtime.

As an example, the program in Figure 2 has the corresponding DFA in Figure 3. In state *DFA #8* (after six occurrences of A) the variable `v` is accessed concurrently (note the outlined nodes), qualifying a nondeterministic behavior in the program, which is refused at compile time.

3.2 Internal events & Dataflow support

Internal events bring dataflow support to the language, allowing that programs create dependency relationships among variables.

Suppose in a program we want that any change to variable `v1` automatically updates `v2` to `v2=v1+1`, and that any change to `v2` updates `v3` to `v3=v2*2`. The program in Figure 4 implements the desired behavior.

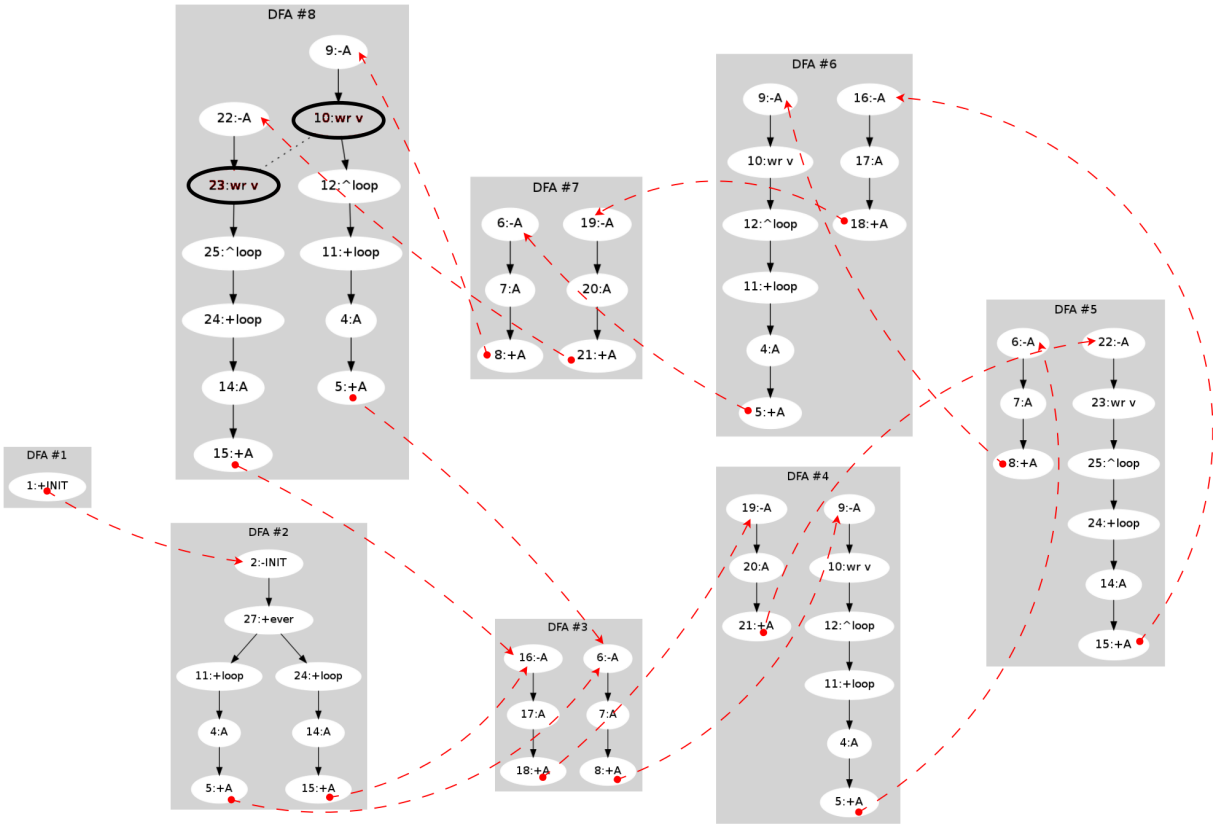

Figure 3: DFA for the nondeterministic example.

We start by defining the variables and corresponding internal events to signal changes (lines 1-2). Any change to a variable in the program must be followed by an emit on the corresponding event so that dependent variables can react. Then, we create two trails, one for each dependency relation, to await for changes and update the variables. For instance, the first trail is a `loop` (lines 4-8) that waits for changes on `v1` (line 5), resets `v2` to apply the constraint (line 6), and signals this change (line 7) to make sure that its dependencies are also updated. The behavior for the second trail (lines 10-14), which updates `v3` whenever `v2` changes, is similar.

3.3 Expected contributions

The goal of our research is to provide safe and high-level programming for embedded systems.

Regarding high-level programming, the first contribution is the unification of control and dataflow reactivity in a single language. Both styles provide

```
 1:  int v1, v2, v3;
 2:  internal void v1_evt, v2_evt, v3_evt;
 3:  par do
 4:     loop do              // 1st trail
 5:        await v1_evt;
 6:        v2 = v1 + 1;
 7:        emit v2_evt;
 8:     end
 9:  with
10:     loop do              // 2nd trail
11:        await v2_evt;
12:        v3 = v2 * 2;
13:        emit v3_evt;
14:     end
15:  with
16:     ...                  // 3rd trail
17:  end
```

Figure 4: A dataflow program.

powerful abstractions for programming embedded systems, but we are not aware of a single language that addresses both styles. We propose a control core similar to the Esterel programming language and include shared state to also support the dataflow style. Another contribution is first-class support for *wall-clock* time (i.e. time from the real world, measured in hours, minutes, etc.), which is probably the most common input in embedded systems, as found in typical patterns like sensor samplings and watchdogs.

Regarding safety, we propose a deterministic execution model in which only a single input event can occur at a time. To contrast with Esterel, in which time is defined as a sequence of *tick* pulses in which multiple events can coexist, our modification enables the proposed static temporal analysis that detects concurrent access to shared state. We also propose a compile-time verification to ensure that loops always run in bounded time.

Finally, we show the viability of our model in the context of embedded systems with an implementation targeting highly constrained platforms.

4 Work plan

Our current activities can be split in three different fronts:

Language design: The overall language design is satisfactory and we do not expect to add new features or dramatically change the current semantics. The next step is to work on a precise description, possibly with the formal operational semantics of the language.

Target platforms: To test the viability of the language in the context of embedded systems, we have been evaluating two embedded platforms. The activity in this area constitutes the integration of the language with the environment, i.e., the definition of input & output events, and the mechanisms for acquisition (e.g. polling vs ISRs).

A challenge is to keep the language runtime with a low memory overhead to permit the development of complex applications. For the platforms we have been working, the runtime is around 10% of the total available memory. We expect to port the runtime to even more constrained microcontrollers, such as the *PIC-16* family.

Application scenarios: In order to evaluate our research goals of providing safe and high-level programming for embedded systems, we intend to apply our language to different application scenarios. Currently, we have been working with wireless sensor networks (WSNs), developing a library of distributed algorithms to be used in networked applications.

We will use quantitative and qualitative metrics in our evaluation. As an example, an existing work [6] measures the performance of different programming languages regarding memory usage, battery consumption, and responsiveness specifically for WSNs. These aspects are of extreme importance, given the severe hardware constraints in this context.

4.1 Schedule

	jul-sep	oct-dec	jan-mar	apr-jun	jul-sep
New platforms	x	x	x		
Language evaluation	x	x	x	x	
- WSNs	x	x			
- New scenarios		x	x		
Language formalization				x	x
Thesis writing				x	x

References

[1] Adya, A., et al. Cooperative task management without manual stack management. In *ATEC '02* (Berkeley, CA, USA, 2002), USENIX Association, pp. 289–302.

[2] Ashcroft, E. A., and Wadge, W. W. Lucid, a nonprocedural language with iteration. *Communications of the ACM 20*, 7 (1977), 519–526.

[3] Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and de Simone, R. The synchronous languages 12 years later. *Proceedings of the IEEE 91* (Jan 2003), 64–83.

[4] Berry, G. *The Esterel-V5 Language Primer*. CMA and Inria, Sophia-Antipolis, France, June 2000. Version 5.10, Release 2.0.

[5] Berry, G., and Gonthier, G. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming 19*, 2 (1992), 87–152.

[6] Duffy, C., Roedig, U., Herbert, J., and Sreenan, C. J. A comprehensive experimental comparison of event driven and multi-threaded sensor node operating systems. *JNW 3*, 3 (2008), 57–70.

[7] Dunkels, A., Schmidt, O., Voigt, T., and Ali, M. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2006), SenSys '06, ACM, pp. 29–42.

[8] Gay, D., Welsh, M., Levis, P., Brewer, E., von Behren, R., and Culler, D. The nesC language: A holistic approach to networked embedded systems. In *PLDI'03* (2003), pp. 1–11.

[9] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE 79* (September 1991), 1305–1320.

[10] Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3 (June 1987), 231–274.

[11] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. System architecture directions for networked sensors. *SIGPLAN Notices 35* (November 2000), 93–104.

[12] Kasten, O., and Rmer, K. Beyond event handlers: Programming wireless sensors with attributed state machines. In *The Fourth International Conference on Information Processing in Sensor Networks (IPSN)* (Los Angeles, USA, Apr. 2005), pp. 45–52.

[13] Moura, A. L. D., and Ierusalimschy, R. Revisiting coroutines. *ACM TOPLAS 31*, 2 (Feb. 2009), 6:1–6:31.

[14] Wan, Z., and Hudak, P. Functional reactive programming from first principles. *SIGPLAN Notices 35*, 5 (2000), 242–252.