LCTES'18 Home

francisco.santanna@gmail.com Profile · Help · Sign out
Response deadline in 12 minutes

#11 A Memory-Bounded, Deterministic and Terminating Semantics for the Synchronous Programming Language Céu

Main Edit Your submissions #16 > (All) Search

Email notification
Select to receive email on updates to reviews and comments.

PC conflicts
None

Submitted

Submission 23 Feb 2018 11:03:32pm EST · 1e269038

▶ Abstract

Céu is a synchronous programming language for embedded soft real-time systems. It focus on control-flow safety features, such as safe shared-memory concurrency and safe abortion of lines of execution, while enforcing memory-bounded, deterministic, and terminating reactions to the environment. In this work, we present a small-step structural operational semantics for Céu and a proof that reactions have the properties enumerated above: that for a given arbitrary timeline of input events, multiple executions

▼ Authors (blind)

Guilherme F. Lima (PUC-Rio) <glima@inf.puc-rio.br>
Rodrigo C. M. Santos (PUC-Rio) <rsantos@inf.puc-rio.br>
Edward Hermann Haeusler (PUC-Rio) <hermann@inf.puc-rio.br>
Roberto Ierusalimschy (PUC-Rio) <roberto@inf.puc-rio.br>
Francisco Sant'Anna (Rio de Janeiro State University) <francisco.santanna@gmail.com>

▶ Topics

	OveMer	RevExp
Review #11A	3	4
Review #11B	4	2
Review #11C	3	2
Review #11D	3	2
Review #11E	4	3

1 review remains outstanding.
You will be emailed if new reviews are submitted or existing reviews are changed.

1 Comment: *Response* (F. Sant'Anna)

You are an **author** of this submission.

Edit submission Edit response

Reviews and comments in plain text

Review #11A

Overall merit

3. Weak accept

Reviewer expertise

4. Expert

Paper summary

The paper presents an operational semantics for the imperative synchronous language Céu (a language which was inspired by Esterel). The semantics is used to prove general properties of determinism, termination (of a reaction) and memory-boundedness.

Comments for author

As a semantic paper, its main section (Section 3, "Formal semantics"), which follows a section in which the language is nicely informally presented, is very technical and not easy to read. This is not necessarily a problem, and indeed, after careful reading, the whole development seems totally correct. However, the reader is not a machine and sometimes, some more explanations or concrete examples for the precise meaning of the rules and intuition behind them could help the reading. One problem of this paper could be the limited space in a conference paper. It would gain probably to be extended in a journal article. A difficulty for the reader is also that the model which is presented is very specific to the language Céu (it probably would not apply to some other formalism). In the description of the operational semantics, a simple running example would help understanding for instance the evolutions of the stack level. Also, the functions described in Figure 4 are barely explained... And a typical example of missing explanations in the subsection on properties is about what is called the "rank" of a description. Here again, this has to be clearly understood to understand the proofs. In addition to these remarks, let's remind the objective of LCTES conference series: "LCTES provides a link between the programming languages and embedded systems engineering communities." So, what about this semantics? How is it used? Is it really used for the implementation of the language? Concerning the language itself, it is clear indeed that the strong restrictions (compared to more general synchronous languages such as Esterel, Lustre or Signal) help to ensure the properties which are proved. However, even if it is the case for other languages (as it is reminded in the "related work" section), the fact of linking the semantics of the language to what could be a particular implementation (scheduling in lexical order) could be considered somewhat questionable. While the language uses, in principle, parallel constructions, the parallelism seems in any case very largely limited by such a choice. Anyway, an interesting feature of the language is the way internal events introduce micro reactions within external reactions. It makes me think, in a different context, to the notion of oversampling in the language Signal, which allows to have internal clocks more frequent than the clocks of input-output signals. Too bad that no specific reference to this language is made here on this subject.

Some additional remarks: line 12: "It focuses" line 573-574: "At the beginning of the reaction": at this point, it is not necessarily clear what reaction it is line 730: missing "e" in bcast lines 841-842: syntactic restrictions regarding loop expressions are not discussed in Section 3 line 859: the fact that * means finite should probably be said in Theorem 3.8 lines 1020-1021: "Theorems 3.14 and 3.14"?

Review #11B

Overall merit
4. Accept

Reviewer expertise
2. Some familiarity

Paper summary
This paper defines formal semantics for the language Ceu, a synchronous, reactive language designed for concurrency and safety. The paper describes the syntactic and semantic features of the language, such as events, reactions, and shared-memory concurrency. It then goes on to describe formal small-step semantics for the language, including properties such as determinism of reactions and finite termination and memory usage. The important result of the paper is that the semantics are deterministic, i.e. given an arbitrary timeline of events, a program will always compute the same result in the same way. Thus, Ceu is a language suitable for distributed applications where safety is important.

Comments for author
What I liked from this paper:

- A formal semantics for a reactive synchronous language Ceu. Importantly, these semantics are fully deterministic, allowing greater safety in the language
- The language features are presented in an easy to understand way, with many demonstrative examples
- The semantics and properties are defined clearly, and presented in a fairly intuitive way. Thus, this paper is easy to understand and follow

Things that could be improved:

- There are no semantics given for certain language features, e.g. what happens when a call to a C function is made using the underscore syntax? Also, it's mentioned that side effects and system calls behave like in conventional imperative languages, and it would be good to have a reference for what this behaviour is.

Typos/Grammar Line 12: focus -> focuses Line 780: The proof of lemma 3.1 -> The proof of lemma 3.2 (maybe check this one)

Questions for authors
Are there approaches to this distributed synchronous programming problem other than the event-driven model?

Review #11C

Overall merit
3. Weak accept

Reviewer expertise
2. Some familiarity

Paper summary
The paper presents a formal semantics for the programming language Ceu.

Comments for author
The technical side of the paper seems sound, but the paper is poorly motivated and provides

little insight how the provided semantics can be used to analyse Ceu programs.

Even for a non-expert like me, the language is well introduced and the formal semantics is well explained. The conclusions are as expected and there is a lack of showing the usefulness of the presented semantics by using it in some setting.

Questions for authors
How does the focus on the control aspects of the language (and leaving out side-effects and system calls) affect the semantics and the conclusions which can be drawn from analysing it?

Review #11D

Updated 21 Mar 2018 5:57:31am EDT

Overall merit
3. Weak accept

Reviewer expertise
2. Some familiarity

Paper summary
The paper extends CEU, a programming language for embedded systems, able to reduce code size by 25% compared to C with events, while increasing the memory overhead by 10%. The extension proposed in this work is a small-step struc- tural operational semantics that guarantees determinism in terms of the final memory state and a bounded reaction time. The extensions are simple (relying on a finalization step and await -synchronous operation) but effective (the guarantees are important in real-time soft systems).

Comments for author
How is determinism ensured across different runs for the example in fig 2a, if events A and B can occur in any order?

It is not clear how many of these semantics exist in the implementation of CEU presented in TECS'17. Neither the introduction, nor the related work describe the novelty of this proposal over the previous publications of CEU. Overall, the paper gives the impression of proposing CEU, since the boundary between the language description and the extensions proposed in this work is not clear (e.g. The bullet point list in the introduction). Is the contribution the actual formalization of these semantics?

Otherwise, how was synchronization between traits implemented previously in CEU to ensure correctness?

Can await-emit lead to a deadlock if called from two concurrent traits, waiting on each other?

How does the compiler ensure that no non-blocking operations are called from CEU? Is there a black list of library calls?

Questions for authors
Explain the contributions w.r.t. the work presented in TECS.

Response

Edit

[Francisco Sant'Anna <francisco.santanna@gmail.com>] 19 Mar 2018 1:09:44pm EDT **784 words**

We would like to thank the reviewers for the thoughtful comments, questions, and criticism.

In a final version, we will address the main concern among reviews regarding the paper motivations. We will discuss why having a formal semantics of the language is important, and how it affected our own implementation. We will also discuss why the main results of the paper (memory boundedness, determinism, and termination) are particularly important in the context of embedded systems.

We will also accommodate all requested clarifications and corrections (e.g., more intuitive/complete descriptions of the semantics, compare with some specific work, etc.). Some of the comments are also addressed individually as follows.

Review #11A

> Concerning the language itself, it is clear indeed that the strong restrictions <...> help to ensure the properties which are proved. However, even if it is the case for other languages <...>, the fact of linking the semantics of the language to what could be a particular implementation (scheduling in lexical order) could be considered somewhat questionable. While the language uses, in principle, parallel constructions, the parallelism seems in any case very largely limited by such a choice.

Considering the constant and low-level interactions with the underlying architecture in embedded systems (e.g., direct port manipulation), we believe that lexical scheduling must be part of the language specification as a pragmatic design decision.

However, Céu has compile-time mechanisms to detect trails in parallel that never share resources (`[Section 2.2]`). In such cases, an implementation could take advantage of real parallelism.

Review #11B

> – There are no semantics given for certain language features, e.g. what happens when a call to a C function is made using the underscore syntax? Also, it's mentioned that side effects and system calls behave like in conventional imperative languages, and it would be good to have a reference for what this behaviour is.

In this context, C calls and system calls are synonymous. The behavior for calls and side effects, all encapsulated in the `mem(id)` primitive, is to modify a memory location in "instantaneous" time. They are opaque operations that have no effect on the application execution flow (e.g., they cannot block or abort execution).

> Are there approaches to this distributed synchronous programming problem other than the event-driven model?

Most synchronous languages have a clock/tick-based model of time (`[Section 4]`). These models are duals since we can accommodate a single event in a single tick and vice versa.

Review #11C

> How does the focus on the control aspects of the language (and leaving out side-effects and system calls) affect the semantics and the conclusions which can be drawn from analysing it?

The focus on control does not affect the logical semantics, abstract program behavior (independent of real-time considerations). Any modeling of side effects would have to assume, or somehow guarantee, not to break the synchronous model (e.g., that system calls are non-blocking). Such assumptions or guarantees would lead us to the same result.

Review #11D

> How is determinism ensured across different runs for the example in fig 2a, if events A and B can occur in any order?

The program behaves deterministically in the sense that the same inputs (event history) will necessarily produce the same output. So, a history where `A` occurs before `B` is different from a history where `B` occurs first. Different inputs can produce different outputs.

> It is not clear how many of these semantics exist in the implementation of CEU presented in TECS'17. <...>. Overall, the paper gives the impression of proposing CEU, <...>. Is the contribution the actual formalization of these semantics?

Yes, the contribution is the actual formalization and proofs of memory boundedness, determinism, and termination. Céu itself has already been presented in other publications such as TECS'17.

> Otherwise, how was synchronization between traits implemented previously in CEU to ensure correctness?

From the very beginning, the language has been designed (and implemented) with these safety properties in mind. The proposed formalization of this paper has helped us to validate the design and has also influenced the current implementation.

> Can await-emit lead to a deadlock if called from two concurrent trails, waiting on each other?

No. An `emit` remains pending until its triggered internal reaction completes. If it is pending it cannot be in an `await`, so a new `emit` in the triggered reaction cannot retrigger the original `emit`. Stack-based internal events prevents code reentrancy that would occur with a queue-based policy.

> How does the compiler ensure that no non-blocking operations are called from CEU? Is there a black list of library calls?

The actual implementation of Céu supports a command-line option that accepts a white list of library calls. If a program tries to use a function not in this list, the compiler raises an error.

Review #11E

Overall merit
4. Accept

Reviewer expertise
3. Knowledgeable

HotCRP