# Macros as Abstractions: Simplifying Code Generation for Lingua Franca

Tassilo Tanneberger
*TU Dresden*
tassilo.tanneberger@tu-dresden.de

Erling R. Jellum
erling.jellum@gmail.com

Jeronimo Castrillon
*TU Dresden*
jeronimo.castrillon@tu-dresden.de

Edward A. Lee
*UC Berkeley*
eal@berkeley.edu

***Abstract*—Current Cyber-Physical Systems (CPS) experience concurrency and compatibility bugs due to manual software integration. Although Lingua Franca's (*LF*) reactor model ensures deterministic execution, which helps make CPS applications safer and more reliable, existing runtimes such as reactor-c struggle to support resource-constrained embedded devices. This paper presents *reactor-uc*'s code generator and macro interface, a lightweight runtime that uses object-oriented C patterns and a macro-based interface to eliminate dynamic memory allocation and simplify code generation. Evaluation results show that *reactor-uc* produces one-third the code size and half the cyclomatic complexity of reactor-c, as well as smaller binaries, while maintaining comparable performance on benchmarks. These improvements significantly enhance the readability and reliability of deterministic software for microcontrollers.**

***Index Terms*—Compiler, Embedded, Code Generation, MoC**

## I. Introduction

Cyber-Physical Systems (CPS) are built with many microcontrollers (MCUs), which perform low-level motor control, read sensors, and communicate with more powerful hardware. Currently, each part of this system is typically a separate software component which must be manually integrated with other components. This approach leads to numerous integration and concurrency bugs. Lingua Franca (*LF*) [1] and *reactor-uc* aim to address these issues by implementing the reactor model of computation (MoC) [2]. The reactor model enables concurrent, even distributed, execution while preserving determinism and providing clear timing semantics. *LF* is a domain-specific language (DSL) for expressing the reactor MoC, and *reactor-uc* is a runtime that implements these semantics and manages execution. Timing and real-time requirements make the CPS field especially challenging, as control input must be supplied in a timely and reliable manner. A model-based approach to control systems has already been popularized due to its better analyzability and provability. The work here leverages lessons learned from reactor-c, which was a first effort to generate efficient, deterministic, concurrent C code using the reactor model. It also leverages lessons learned from reactor-cpp, which focuses on clean object-oriented design. Our work makes several improvements that are particularly beneficial for programming embedded systems, including avoiding dynamic memory allocation and deallocation, leveraging a wider variety of networking and communication mechanisms, and generating more compact and readable code.

Section II explains the *LF* language and the reactor MoC. Next, Section III provides a brief introduction of *reactor-uc* and its design philosophy. Our primary contributions are presented in Section III.D and Section III.E, where we describe our code generator and macro interface. In Section IV, we assess our code generation approach. Finally, future ideas and directions are discussed in Section V.

## II. Background

### A. Reactor Model of Computation

The reactor model of computation [2] combines concepts from many other MoCs, such as the actor model [3], logical execution time (LET) [4], and synchronous reactive languages like Esterel [5]. Reactor programs consist of reactors, which are stateful components similar to actors. In comparison, the communication between reactors happens through ports, making them more modular, where actors directly refer to each other. Reactors can also contain components not present in the actor model, such as timers and logical actions. The business logic is encapsulated in reactions, which are numbered based on their order of declaration. This numbering ensures that no two reactions inside the same reactor are executed concurrently, which would introduce nondeterminism into the model. Timers trigger periodically, while logical actions can be scheduled for a specific time in the future to trigger reactions, while physical actions are used to introduce asynchrony and nondeterminism into the model. The reactor model generalizes the LET principle [4], differentiating between logical time and physical time. Because *LF* and the reactor model are declarative in nature, by default the logical execution time of a reaction is zero, so lag (logical-to-physical delay) can be understood as a deviation from the specification.

### B. Lingua Franca

*LF* [1] is a declarative DSL that expresses the reactor MoC previously introduced. In *LF*, reactors and their topologies are defined. Code generation, the focus of this paper, is also a widely adopted method across different MoCs; for example, Esterel is also translated into C. Many *LF* runtimes have been developed to support different target languages; the most

notable are the reactor-c and reactor-cpp [6] runtimes. The static scheduling methodology [7] is of particular interest for CPS applications. It translates *LF* programs into a quasi-static schedule. This static scheduler replaces the dynamic scheduler and offers beneficial real-time properties, such as improved timing precision, and can be used to verify that deadlines are met given WCETs on reactions. The static scheduling methodology has also been used to optimize for other criteria, such as energy efficiency [8] on heterogeneous multi-cores.

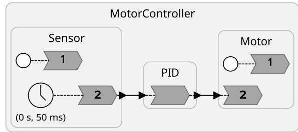



Fig. 1. Controller for a brushless DC motor

*LF* can now be leveraged to build control systems. A classical control loop would not be defined with an imperative loop, but with a timer driving a chain of reactions periodically. This is illustrated in Fig. 1, where a timer inside the Sensor reactor triggers a first reaction responsible for fetching a sensor value (e.g., velocity) and then passing this sensor reading to the controller, which in turn forwards its control output to the motor. Because of the model-based and declarative nature of *LF*, we can reason about and verify this program more easily. Especially for control systems, the ability to reason about time is critical. This makes it possible to provide assurances about the program before it reaches the production environment.

## III. reactor-uc Compile Flow

This section introduces *reactor-uc*, its macro interface, and the code generator that produces code following this interface. *reactor-uc* is a runtime that facilitates the execution of a program when it is fed a reactor program translated to C. The code generator takes an *LF* program and translates it into C code, which is then compiled and linked with the runtime to create a binary that can be flashed onto a microcontroller. For this process to work, the code generator and runtime must agree on an interface to validly translate *LF* programs that *reactor-uc* can understand. This is the focus of this paper and this section, as it deals with *reactor-uc*'s macro interface.

### A. The reactor-uc Runtime

*reactor-uc* is a new runtime for *LF* designed for resource-constrained embedded devices. To this end, *reactor-uc* targets single-core platforms and supports many real-time operating systems (RTOSes), such as RIOT[9], Zephyr, FreeRTOS, and various SDKs like the Raspberry Pi pico-sdk. The design of the *reactor-uc* runtime is based on four core principles: 1. extensibility, 2. embedded as a first-class citizen, 3. ease of integration, and 4. human understandability. Extensibility means *reactor-uc* provides interfaces for implementing new platforms or schedulers. Embedded as a first-class citizen means that *reactor-uc* is optimized for single cores and does not perform any dynamic memory allocation. For ease of integration, *reactor-uc* generates CMake and Makefile files, which can be integrated with platform-specific tooling. These files are used by template repositories for different platforms. *reactor-uc* configuration occurs at compile time via compile flags to select the platform or scheduler used. Lastly, human understandability is achieved through clear abstractions and hierarchies (see Section III.C), as well as by making the generated code more human-readable, as explained in Section III.D.

### B. No Dynamic Memory Allocation

In *reactor-uc*, reactor components like ports, actions, timers, and connections are all triggers, and triggers require lists of triggers, observers, and sources. These arrays are provided during the initialization of the elements; they must be code-generated and then passed in, as described in more detail at Section III.D. This pre-allocation is also necessary for value buffers in actions, delayed connections, and for the event and reaction queues. This increases the complexity within the code generator, because now every action, every downstream element, and every trigger must be carefully counted. There are two major benefits to this approach. First, the amount of memory needed is known at compile time, which is very beneficial for constrained devices where memory is a scarce commodity. Second, supporting new embedded or bare-metal platforms becomes easier because the *reactor-uc* platform model does not need to supply a `malloc` abstraction.

### C. Object Orientation in C

It is a common technique in C programming that by nesting `structs` at the first position and ensuring proper memory alignment, that objects can be freely cast between them. In *reactor-uc*, this technique is heavily used. For example, a timer "inherits" from the Trigger class by including an instance of the Triggers `struct` as the first element of the timer `struct`. This allows a trigger to be cast to a timer and vice versa. To achieve polymorphism, the base-`structs` carry function pointers e.g. `cleanup` or `prepare`, which are overwritten by the sub-class upon initialization of the object.

### D. Macros

C macros are notoriously difficult to use correctly, but they can make implementing code generators for C much easier. *reactor-uc* exposes a set of macros to define an interface that the code generator uses, while the underlying implementation can change without modifying the code generator. This is especially useful when lowering MoCs into C; components of the reactor MoC, such as Timers, can be instantiated with just a few macro statements. For every reactor, the *reactor-uc* code generator creates a header file and a C file. The header file contains all `struct` definitions, including the `struct` for the reactor itself, and declares the reactor constructor. In contrast, the C file contains only the reaction bodies and constructor implementations for all components and the reactor itself. Listing 1 shows how a timer is translated into C code. This macro takes four arguments: first, the `ReactorName`, which is necessary for scoping; second, the `TimerName`, which, together with the `ReactorName`, specifies the name of the `struct`; third, the `EffectSize`, which specifies how many reactions are triggered by the timer; and fourth, how

many reactions can observe the timer but are not triggered by it.

```c
#define LF_DEFINE_TIMER_STRUCT(ReactorName, TimerName, EffectSize, ObserversSize) \
  typedef struct { \
    Timer super; \
    Reaction* effects[(EffectSize)]; \
    Reaction* observers[(ObserversSize)]; \
  } ReactorName##_##TimerName;

#define LF_DEFINE_TIMER_CTOR(ReactorName, TimerName, EffectSize, ObserverSize) \
  void ReactorName##_##TimerName##_ctor( \
    ReactorName##_##TimerName* self, \
    Reactor* parent, interval_t offset, \
    interval_t period) { \
    Timer_ctor(&self->super, parent, offset,\
      period, self->effects, EffectSize, \
      self->observers, ObserverSize); \
  }
```

Listing 1. Definition of the `LF_DEFINE_TIMER_STRUCT` and `LF_DEFINE_TIMER_CTOR` macro.

For each reactor component, we generate four distinct elements: the `STRUCT`, the constructor `CTOR`, the instantiation `INSTANCE`, and the initialization `INITIALIZE`. Each reactor component declaration is translated into its own `struct`. This means that if a reactor has two timers, the generator would generate two timer structs. This is necessary because the number of effects, observers, and sources may vary between the different declarations of the same type.
The `LF_TIMER_INSTANCE` macro adds an instance of that timer to the reactor `struct` (see Listing 2). The `INITIALIZE` macro creates an invocation to the constructor generated by the `CTOR` macro. This call occurs inside the reactor's constructor (see Listing 3). These constructors call the constructors of their respective base classes to initialize the class hierarchy. This is shown in line 13 in Listing 1. The constructor generated by the `CTOR` macro passes the generated effects, observer, and sources pointers into the base class constructor. This makes the generated code self-contained.

```c
#define LF_TIMER_INSTANCE(ReactorName, TimerName) ReactorName##_##TimerName TimerName

#define LF_INITIALIZE_TIMER(ReactorName, TimerName, Offset, Period)                    \
  self->_triggers[_triggers_idx++] = (Trigger*)&self->TimerName; \
  ReactorName##_##TimerName##_ctor(&self->TimerName, &self->super, Offset, Period)
```

Listing 2. Definition of the `LF_TIMER_INSTANCE` and `LF_INITIALIZE_TIMER` macro.

An advantage of using macros is that naming scheme of `structs` and constructors is completely defined and abstracted by the macros. The code generator does not need to know about the naming, but only needs to make sure the correct arguments are passed into the macros. This helps avoid bugs inside the code generator, where inconsistent names may be generated. Output ports are an exception because they cannot be self-contained; output ports require an array of outgoing connections. The number of connections depends on where the reactor is instantiated. Therefore, the containing reactor, where the connections are declared, needs to allocate an array of connections. All of these macros are technically standalone and could also be used by humans to express reactor programs, but correct counting of objects and calculating sizes is critical, so it is recommended to use our *LF* code generator introduced in Section III.E.

### *E. The Code Generator*

The *reactor-uc* code generator is primarily written in Kotlin, where the compiler frontend is built in XText[1]. For all reactor components, there are generator classes, such as `UcTimerGenerator`, which takes an object representing a reactor. Then this class generates, for every timer instantiation in the reactor, the corresponding `struct`, constructor, instantiation, and initialization. This process is similar for all other reactor components. All these generator classes expose the methods `generateSelfStructs`, `generateCtor`, `generateStructField`, and `generateCtorCode`, which produce the four essential macros previously introduced.

```kotlin
private fun generateReactorStruct() = with(PrependOperator) {
        """ |typedef struct {
            |  Reactor super;
        ${" |  "..instances.generateReactorStructFields()}
        ${" |  "..reactions.generateReactorStructFields()}
        ${" |  "..timers.generateReactorStructFields()}
        ${" |  "..actions.generateReactorStructFields()}
        ${" |  "..connections.generateReactorStructFields()}
        ${" |  "..ports.generateReactorStructFields()}
        ${" |  "..state.generateReactorStructFields()}
        ${" |  "..parameters.generateReactorStructFields()}
            |  LF_REACTOR_BOOKKEEPING_INSTANCES(${reactor.allReactions.size}, ${numTriggers()}, ${numChildren});
            |} ${reactor.codeType};
            |""".trimMargin() }
```

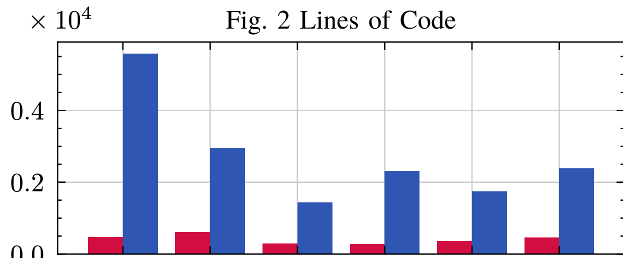
Listing 3. Sequence of the code generator, producing the reactor `struct`

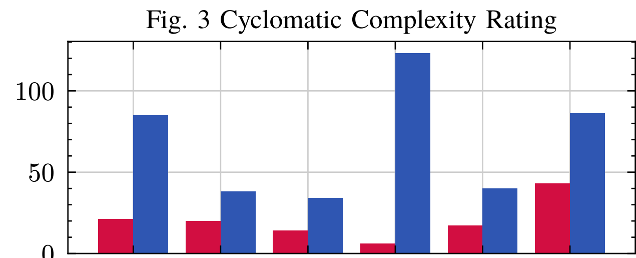
In Listing 3, the sequence shows the generation of a reactor `struct`. Notice that the generic `Reactor` struct is the first element instantiated, aligning with the approach presented in Section III.C. Additionally, the `generateReactorStructFields` method generates all the instantiations of the reactor components.

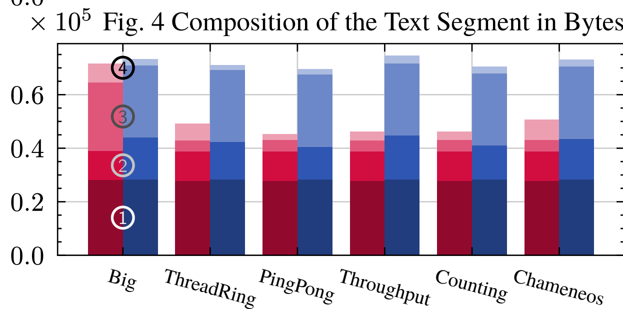
## IV. Evaluation

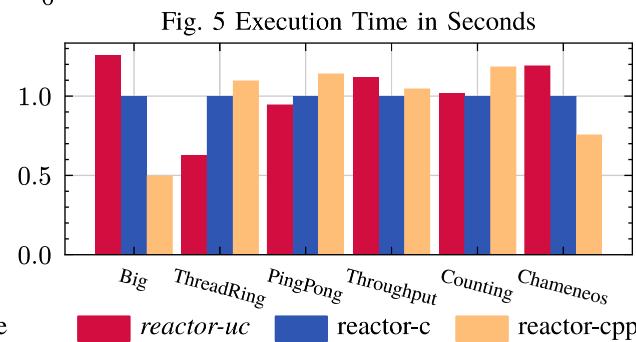
To evaluate the *reactor-uc* approach, we compare it with reactor-c, a mature *LF* runtime implementation. reactor-c serves as a good baseline because it uses the same MoC and generates code in the same target language (C). The comparison focuses on the generated code. As indicators of generated code quality, this paper examines the number of lines of code generated and code complexity. These metrics are measured by SCC[2] (Sloc Cloc and Code), a widely used tool for code analysis and linting. The complexity rating produced is cyclomatic complexity [10], which models human understandability. The size of the Text segment is also measured. The compiled binary uses the pico-sdk for the RP2040, a widely adopted microcontroller. We also perform limited performance measurements. We test on the micro-benchmarks

[1]https://eclipse.dev/Xtext/
[2]https://github.com/boyter/scc




from the Savina benchmark suite [11]. The results in Fig. 2 show that reactor-c's code generator produces roughly three times more code than *reactor-uc*'s code generator for the same benchmarks. This is unexpected because *reactor-uc* incurs additional overhead from pre-allocating everything. The complexity rating (see Fig. 3) of *reactor-uc* is also less than half that of reactor-c, indicating that the code is more understandable. reactor-uc produces smaller binaries than reactor-c. The pico-sdk memory footprint is very similar for both runtimes, while the reactor-uc runtime implementation is slightly smaller. Furthermore, it uses fewer newlib (embedded libc) functions, which accounts for the largest difference. The complexity rating difference are mainly due to reactor-c's very complicated initialization process. Here, reactor-c generates one large function that initializes every reactor and its components in the entire program; additionally, all dependencies between components are encoded within this single function. Regarding performance results (shown in Fig. 5), *reactor-uc* and reactor-c are very similar.

## V. Future Work

*reactor-uc* is still in development and has not released an initial version, so this system may change. One proposed idea is to allocate a very large array statically at the beginning and pass only this single array into the *reactor-uc* runtime. Then, *reactor-uc* could provide its own `malloc` implementation, similar to an OS, using the large pre-allocated array. This would remove much of the complexity from the macro definitions and require only a very simple `malloc` implementation inside the runtime. The benefit is that issues like fragmentation are avoided because memory is never freed; thus, `malloc` could segment the memory linearly based on the requested size. Furthermore, this approach can also be leveraged to simplify static-schedule instruction generation [7], where instructions are represented by macros. In general this is good evidence that this design can be leveraged by many declarative languages expressing MoCs.

## Acknowledgement

This works was supported, in part, by the EU Horizon Europe Programme under grant agreement No 101135183 (MYRTUS) and by the German Research Council (DFG) through the REC$^2$ Excellence Cluster (EXC 3035).

## References

[1] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, "Toward a Lingua Franca for Deterministic Concurrent Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 4, May 2021, doi: 10.1145/3448128.

[2] M. Lohstroh *et al.*, "Reactors: A Deterministic Model for Composable Reactive Systems," in *Cyber Physical Systems. Model-Based Design*, Cham: Springer International Publishing, 2020, pp. 59–85.

[3] G. A. Agha, P. Thati, and R. Ziaei, "Actors: a model for reasoning about open distributed systems," in *FME 2001*, 2001. [Online]. Available: https://api.semanticscholar.org/CorpusID:202302

[4] C. M. Kirsch and A. Sokolova, "The Logical Execution Time Paradigm," in *Advances in Real-Time Systems*, S. Chakraborty and J. Eberspächer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 103–120. doi: 10.1007/978-3-642-24349-3_5.

[5] F. Boussinot and R. de Simone, "The ESTEREL language," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293–1304, 1991, doi: 10.1109/5.97299.

[6] C. Menard *et al.*, "High-performance Deterministic Concurrency Using Lingua Franca," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 4, Oct. 2023, doi: 10.1145/3617687.

[7] S. Lin *et al.*, "Quasi-Static Scheduling for Deterministic Timed Concurrent Models on Multi-Core Hardware," *ACM Trans. Embed. Comput. Syst.*, vol. 24, no. 5s, Sept. 2025, doi: 10.1145/3762653.

[8] S. Lin *et al.*, "Navigating Time and Energy Tradeoffs in Reactive Heterogeneous Systems," *IEEE Embedded Systems Letters*, vol. 17, no. 2, pp. 103–106, 2025, doi: 10.1109/LES.2024.3469278.

[9] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2013, pp. 79–80. doi: 10.1109/INFCOMW.2013.6970748.

[10] D. Wallace, A. Watson, and T. Mccabe, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric." Special Publication (NIST SP), National Institute of Standards, Technology, Gaithersburg, MD, 1996.

[11] S. M. Imam and V. Sarkar, "Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries," in *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, in AGERE! '14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 67–80. doi: 10.1145/2687357.2687368.