Composing Real-Time Applications from Communicating Black-Box Components

Martin Becker[1], Alejandro Masrur[2], and Samarjit Chakraborty[1]
[1]Institute for Real-Time Computer Systems, TU Munich
[2]Software Technology for Embedded Systems, TU Chemnitz

Abstract— To handle complexity, embedded software is usually divided into components that are developed independently from each other and then need to be integrated in a reliable and deterministic manner. This involves buffering and synchronizing exchanged signals, as well as finding a feasible execution schedule, which is a tedious and error-prone procedure. We propose a model of computation that enables a programming framework which automatically performs such an integration, without requiring access to the components' source code. The developer only needs to declare interface signals between the components, connect them and define their execution periods. A software library then *synthesizes* deterministic communication mechanisms and provides a flexible, yet safe interface for time-triggered execution. Our approach does not require any run-time environment or special compiler, which makes it light-weight and amenable to be used on embedded platforms with limited resources.

I. Introduction

Embedded software is typically structured into a number of components that can be programmed, tested and debugged independently from each other. For example, a signal processing application might comprise of sensor tasks, processing tasks and actuator tasks, all running at different speeds but are supposed to interact with each other. This requires defining *interfaces* between them, i.e., capturing which data is exchanged and at which points in time. Naturally, each component assumes that its interface is satisfied by all other components. As a result, when integrating components into one *composite* program, it needs to be guaranteed that the assumptions made at the individual components can be fulfilled. Towards this, it becomes essential to find a suitable execution schedule and to manage the communication between components. This involves configuring precise points in time at which components are executed, as well as instantiating and synchronizing communication buffers between them.

In this paper, we are concerned with *black-box* components given in the form of *garbled* C code or object code, that shall be executed periodically at different rates, and (logically) in parallel (see Fig. 1). We assume that only the components' interfaces and execution times are known and that the developer has no precise control over the scheduling of the components, which are common constraints from many industrial contexts. For such a setting, we propose an approach that automatically performs a safe and efficient integration of the components as explained before.

Our Contribution: First, we present a model of computation supporting periodic and parallel execution of black-box software components, without making any assumptions on the target platform such as, for example, on the operating system or the scheduler. In this model the components run at different rates and communicate with each

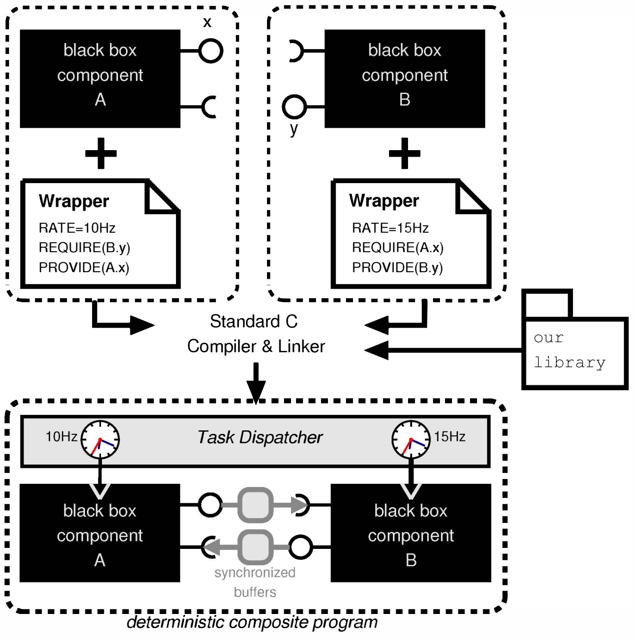

Fig. 1. Workflow of our approach.

other through *signals* and *events*. Based on this model, we then analyze properties that are relevant for our composition, e.g., bounded communication delays. We also show how these properties can be verified on a target platform that implements the model.

Second, to address the difficulties that arise when composing applications from communicating components, e.g., buffering and synchronizing signals between components and managing their time-triggered execution, we provide a C library that follows our model, carrying out such work automatically. The library allows specifying the components and their interfaces in the C language itself, in an intuitive and natural way, without needing to know all the components in the system. The user is only required to declare inputs and outputs of the individual components and to connect them to the black-box. The conveyed information is leveraged during the build process of a standard C compiler/linker, where communication and synchronization mechanisms are then generated automatically.

The rest of this paper is organized as follows. After giving an overview of related work, we introduce our model of computation in Section 3. In Section 4 we analyze our model regarding its real-time properties, which we built upon later. Section 5 then describes the software library, how it extends the C language with constructs for declaring the components and how it ensures that all interfaces are satisfied. Finally, the last section presents a case study illustrating the usefulness of the proposed library and evaluating its incurred memory and execution overhead.

Authorized licensed use limited to: UNIVERSITAETSBIBLIOTHEK CHEMNITZ. Downloaded on April 13,2022 at 15:12:51 UTC from IEEE Xplore. Restrictions apply.

II. Related Work

Approaches like *Nemo* [2], *42* [8], Architecture Analysis & Design Language (AADL) [5], *Timed Multitasking* (TM) [7] (later extended by *COMDES* [1]) and *PTIDES* [4] support integrating software components and also automatic code generation for deploying them. They introduce elaborate languages for defining hierarchical entities, interface contracts, constraints, resource awareness, etc. In general, all these approaches have the disadvantage that components have to be developed based on their respective programming constructs. This may not only cause a significant effort, in particular, when code is already existing, but in contrast to our approach, an integration of black-box components is not possible.

A more light-weight programming framework is *Giotto* [6], which integrates black-box components given in C or Java in a time-triggered multi-rate setting. Similar to Giotto, we are not constrained to a particular scheduling scheme and we have a non-zero communication delay between components. A difference is, however, that Giotto defines its own language, which requires an additional compiler and a run-time environment (interpreter). As a result, in comparison to our approach, Giotto incurs a larger memory overhead, as well as a rather steep learning curve for the language itself.

Port-based objects (PBOs) [10] treat components as black boxes, and focus on their deterministic composition under multi-threading. In particular, PBOs have some similarities to our approach, focusing on simplicity and determinism in a multi-rate setting. However, their components are not synchronized to each other and also require a more complex syntax.

Finally, in contrast to some of the mentioned related works, our approach does not require separate compilers ([1, 2, 3]) or run-time environments ([7]). In particular, some run-time environments impose demanding platform requirements and are oftentimes not available for small embedded platforms (e.g., [8], the Java Virtual Machine) typically used in cost-sensitive domains.

III. Model of Computation

In this section we introduce the definition of and requirements for the black-box components, as well as the details of execution and communication semantics.

A. Definition of Black Box

We define a *black box* as a software component, from which only its *interface* and its high-level behavior are known. It implements some functionality and – together with other black boxes – composes an overall application.

A black-box component can be given either as a library, as object code or even as C source code. In addition, each black-box component has to comply with the following requirements:

(1) The component performs a time-discrete computational task which takes place periodically in time steps of length T_i.

(2) The component has to advance *logical* time (i.e., the time from the component's perspective) by T_i at every invocation.

(3) The component must have a bounded worst-case response time (WCRT) in combination with all other components, i.e., no infinite loops and no indefinitely blocking calls are allowed.

The WCRT of a component depends on the target platform and other implementation details. We illustrate how to obtain the WCRT of a component in Section 4B.

The concept of periodic and discrete computation is common and can be found in signal processing programs, such as those obtained from Simulink models, but also in many other contexts. As a result, our model does not dictate the programming language used for implementing the individual components, which is a major distinction to other approaches from the literature.

B. Model of Execution

As a basic programming entity in our model, we introduce a *container*. A container encloses one black-box component and runs logically in parallel to other containers. It provides an interface to the component, which automatically manages the periodic execution as well as synchronization of communication.

Clocks: Let us denote by C_i any component in the system, where $i = 1 \ldots n_c$ and n_c is the number of components. They are triggered periodically, with a direct relation to *physical* time. Towards this, there exists a *basic clock* with a period T_b, that "ticks" at the time instants $t = k \cdot T_b$, with $k \in \mathbb{Z}$ indexing the ticks. Then, each component C_i shall be executed with an *individual* period T_i, which must be an integer multiple of T_b:

$$T_i = \rho_i T_b,\ \rho_i \in \mathbb{Z}^+, \tag{1}$$

We will refer to ρ_i as *prescaler* of container C_i.

Execution Sequence: A component C_i is triggered every T_i time units, where the following three steps take place:

1. **Read-in (R):** all required inputs are obtained from other components and get buffered.
2. **Process (P):** the black-box component is executed on the buffered inputs and produces buffered outputs.
3. **Write-back (WB):** the outputs are propagated from their buffer to other components.

Table I depicts the relation of these three steps to physical time, where the $W(\cdot)$-operator means the *Worst-Case Response Time*, as explained in the next section.

TABLE I
Execution sequence of a container C_i triggered at time kT_i, with associated logical and physical time.

step	logical time	executed anywhere in physical time interval	
read-in (R)	kT_i	kT_i+	$(0\ ,\ W(\mathrm{R})]$
process (P)	kT_i	$kT_i+W(\mathrm{R})+$	$(0\ ,\ W(C_i)]$
write-back (WB)	$(k+1)T_i$	$kT_i+W(\mathrm{R})+W(C_i)+$	$(0\ ,\ W(\mathrm{WB})]$

The steps *read-in* and *write-back* serve as *synchronization points* for the communication. That is, independently of when the inputs or outputs are getting produced, they are only propagated between the components at these steps. Through this, we are free to execute C_i at any time within its period without running into non-determinism under preemptive scheduling, unlike other approaches based on buffered communication such as [9].

Authorized licensed use limited to: UNIVERSITAETSBIBLIOTHEK CHEMNITZ. Downloaded on April 13,2022 at 15:12:51 UTC from IEEE Xplore. Restrictions apply.

C. Model of Communication

In this section we discuss how data exchange between components takes place.

Communication: Components can provide or receive two basic types of data: events and signals.

1. **Events** are boolean data items which can either be *present* or *absent* , whereas the latter is the default state. Events are *registered.* That is, once an event is present, it remains present, until the sink (i.e., the receiving component) has read or retrieved it.
2. **Signals** are data items which carry a value and are always *present.* They are *sampled* by the sinks. That is, if a component C_i is released every T_i time, it can only obtain the freshest value of a signal at exactly these time instants.

Both events and signals are *directed.* They have exactly one source component and one or multiple sink components. However, without loss of generality, signals/events with multiple sinks can be regarded as multiple point-to-point signals/events.

Signals or events can be emitted by a component C_i at any physical time t. However, from the perspective of logical time, they are regarded as being emitted at time $\lfloor \frac{t}{T_i} \rfloor T_i$, i.e., at the *beginning* of the component's period, as if the component would finish computation in zero time.

In order to obtain a deterministic behavior, signals and events cannot be read by a sink as long as the source is possibly still running. Hence, one has to account for the actually non-zero execution time of the components. Towards this, we assume that an emitted signal or event is not available earlier than $\lceil \frac{t}{T_i} \rceil T_i$, i.e., at the *end* of the component's period. Clearly, this requires that the components finish their computation before their next invocation time, but also introduces a *communication delay* between any two components.

IV. Anaysis of the Model

In the following we show that the chosen model of computation exhibits properties which are beneficial for real-time systems, viz., that the communication delay between the components is bounded and that the semantic correctness of an implementation on a specific target platform can be verified with standard methods.

A. Communication Delay

In the following we give analytical bounds on the *minimum* delay, which plays an essential role in verifying the correctness of an implementation, as well on the *maximum* delay, which defines the freshness of signals and events, i.e., is vital for system performance.

We define the *communication delay* as the latency between the logical emission of a signal or event at the source component and its logical arrival at the sink component.

Example: The communication delay between two components is illustrated in Fig. 2: A component C_0 with $\rho_0 = 3$ *logically* emits an event at time $t = 0$. C_0 is triggered at $t = 0$ and finishes computation anywhere in $[0, \rho_0 T_b[$. Although this means that the event is *physically* produced anywhere in $[0, \rho_0 T_b[$, it is not available to other components before $\rho_0 T_b = 3T_b$.

A second component C_1 with $\rho_1 = 1$ receives this event at time $t = 3T_b$ and immediately echoes it back. The echo is logically emitted at $t = 3T_b$, but not available until time $t = 4T_b$. C_0 retrieves the echo at time $t = 6T_b$. The communication delay is hence $3T_b$ in both directions.

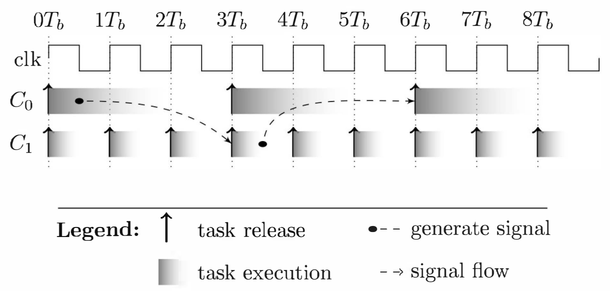

Fig. 2. Illustration of communication delay for our echo example: C_0 sends an event, and C_1 responds as soon as possible.

Corollary 1. ***Minimum Delay.*** *Given a source C_{src} and a sink C_{snk}, the minimum communication delay of an event or signal from C_{src} to C_{snk} is $D^{min}_{src,snk} = T_{src}$ where T_{src} is the activation period of C_{src}.*

Corollary 1 follows from the earlier definition that an event or a signal emitted by C_{src} at a given time t is not available to any C_{snk} before $t + T_{\text{src}}$.

Theorem 1. ***Maximum Delay of Events.*** *The maximum communication delay of an event from a component C_{src} to a component C_{snk} is given by $D^{max,evt}_{src,snk} = T_{src} + T_{snk} - GCD$, where GCD is the* greatest common divisor *of the activation periods T_{src} and T_{snk}.*

Theorem 1 follows from observing that in the worst case C_{snk} might be released just before an event (that was sent by C_{src} at time t) becomes available, i.e., some time δ before $t + T_{\text{src}}$. In this case, C_{snk} *sees* this event at $t + T_{\text{src}} - \delta + T_{\text{snk}}$. Clearly, the smaller δ, the larger the communication delay of this event. Using Bézout's Identity [11, p.45, 87], it can be concluded that the smallest possible δ is given by the greatest common divisor (GCD) of T_{src} and T_{snk}. The theorem follows.

Theorem 2. ***Maximum Delay of Signals.*** *The maximum communication delay of a signal from a component C_{src} to a component C_{snk} is given by $D^{max,sig}_{src,snk} = T_{src} + \min\{T_{snk}, T_{src}\} - GCD$, where GCD is the* greatest common divisor *of the activation periods T_{src} and T_{snk}.*

Theorem 2 is a slight variation of Theorem 1. Depending on the values of T_{src} and T_{snk}, signals can be *overwritten* by C_{src} before C_{snk} sees the last change. As a result, we obtain that $D^{\text{max,sig}}_{\text{src,snk}} \leq D^{\text{max,evt}}_{\text{src,snk}}$.

B. Correctness of the Deployed System

So far we discussed the semantics of our model and derived analytical bounds for the communication delay. Next we sketch how to verify whether these properties hold true for an implementation on a particular target platform.

Worst-Case Execution Time Analysis: As a first step, the *worst-case execution time* (WCET, i.e., the maximum computation time, without considering interruptions from other processes) e_i of the components C_i, as well as for the operations *read-in* and *write-back* has to be found. At this point platform-specific details, such as operating system, caches, etc. have to be considered.

Worst-Case Response Time Analysis: As second step, a worst-case response time (WCRT) analysis needs to be performed for each component C_i based on the WCETs e_i, the periods T_i and the used scheduling policy.

Authorized licensed use limited to: UNIVERSITAETSBIBLIOTHEK CHEMNITZ. Downloaded on April 13,2022 at 15:12:51 UTC from IEEE Xplore. Restrictions apply.

Verification of Semantic Correctness: As a last step, we need to verify whether the implementation on a specific processor follows the semantics. For this, the following condition needs to hold for each component C_i:

$$W(\mathrm{R}) + W(C_i) + \mathrm{W}(\mathrm{WB}) < \min_{j\in \mathrm{sink}(i)} \{D_{i,j}^{\min}\} \quad (2)$$

where $W(C_i)$ is the result of the previously calculated values from the WCRT analysis, sink(i) denotes the set of all indices j where C_j is a sink for any signal/event produced by C_i, and $D_{i,j}^{\min}$ is the minimum communication delay from Corollary 1. The condition results from the latest possible point in time until when a component has to *physically* produce a signal/event, such that it *logically* makes no difference for all of its sinks, i.e., such that it does not break the communication semantics.

However, from Corollary 1 it follows that $\min_{j\in \mathrm{sink}(i)} \{D_{i,j}^{\min}\} \equiv T_i$, thus eq. (2) becomes:

$$W(\mathrm{R}) + W(C_i) + W(\mathrm{WB}) < T_i. \quad (3)$$

Equation (3) allows to verify whether properties such as the maximum communication delay are preserved by an implementation, without the need to know how the components are interconnected, which is a considerable advantage of our model.

V. The C Library

We developed a C library which follows the models introduced in the previous sections. The goal of this implementation was to show that your model could be realized easily and proves useful in a practical setting. Although this is not the best possible implementation, we also provide some measurements on the incurred overhead.

The library introduces new, mainly declarative, constructs to C, which are used to declare the components, their interfaces (signals and events), and to invoke the black boxes. This way, the user does not have to be concerned about implementation details and benefits from an intuitive syntax and high readability. The library exhibits properties suitable for real-time systems:

- small, static memory footprint,
- support for multi-threading,
- provision of signal/event trace logging, and
- any non-deterministic use is prevented (program does not compile and link if there is no driver, multiple drivers, non-matching data type or unsynchronized signals/events).

It works with any standard C89 compiler/linker, and thus can be used in a wide range of applications. The declarative constructs shown in the next subsections are realized through C pre-processor macros.

A. Declaring Components

Each black-box component has to be declared separately in its own *translation unit*. These are later synthesized to containers (see § B), which provide the necessary communication mechanisms. Towards this, the following functions need to be implemented by the user: [1]

- `static void CONT_READIN_ME(void)`: For each signal that is an input to the component, the macro `UPDATE_SIGNAL` has to be called, which synchronizes the signal.

[1]Note that code for *write-back* is generated automatically and needs not be written by the user.

```
// file c0.c, component 0
#define CONT_NAME C0
CONT_PRESCALER(3);

ANNOUNCE_EVENT(echo_1);
REQUIRE_EVENT(echo_2);

void CONT_READIN_ME() {
  UPDATE_EVENT(echo_2);
}

void CONT_PROCESS_ME() {
  SET_EVENT(echo_1);
}
//
```

```
// file c1.c, component 1
#define CONT_NAME C1
CONT_PRESCALER(1);

ANNOUNCE_EVENT(echo_2);
REQUIRE_EVENT(echo_1);

void CONT_READIN_ME() {
  UPDATE_EVENT(echo_1);
}

void CONT_PROCESS_ME() {
  if (GET_EVENT(echo_1))
    SET_EVENT(echo_2);
}
```

Fig. 3. The echo example from Fig. 2 realized with our library.

- `static void CONT_PROCESS_ME(void)`: This function is invoked every T_i time units. The user has to do the necessary bits to invoke the black-box component, and get the results from it. Afterwards, the macro `SET_SIGNAL` has to be called for every signal that is an output of the component.

To illustrate the usage, the complete code that implements the echo example is given in Fig. 3.

B. Synthesizing Communication Mechanisms

The communication buffers for signals and events are generated automatically and kept hidden from the user. These are created when one of the following macros is used:

- `ANNOUNCE`: Indicates that the component associated with the current translation unit produces an output signal/event with the given name and type, which can be used by other components.
- `REQUIRE`: Indicates that the component associated with the current translation unit expects that a signal/event with the given type and name exists.

Each of these macros instantiates a part of a guarded double buffer inside the corresponding translation unit, which works in the following way: Source containers hold one part of a double buffer for each signal/event, which gets instantiated with the macro `ANNOUNCE`. The sinks hold the other part of the double buffer – instantiated with the macro `REQUIRE` – and use the macro `UPDATE` to read from the source's part. This macro invokes a callback to the corresponding *reading function* that is provided by the source (also through `ANNOUNCE`). These functions only allow for a copy operation when the source container is not possibly running. The other macros `SET` and `GET` only work on the local part of the double buffer.

Due to this arrangement, the overhead for writing a signal/event at the source side is negligible, whereas the reading introduces a synchronization overhead. However, in the following we show how this overhead can be reduced further. An optimization of the write phase could be done analogously, but does not have much impact due to the aforementioned reasons.

B.1. Reducing Synchronization Overhead

If the source runs slower than the sink ($T_{\mathrm{src}} > T_{\mathrm{snk}}$), it frequently happens that the sink tries to read a signal or event from the source which has not been updated yet (from Corollary 1: $D_{\mathrm{src,snk}}^{\min} = T_{\mathrm{src}}$ and $T_{\mathrm{src}} > T_{\mathrm{snk}}$). In this case, we can skip the reading process and hence *no synchronization* is necessary then. It can be shown that this

Authorized licensed use limited to: UNIVERSITAETSBIBLIOTHEK CHEMNITZ. Downloaded on April 13,2022 at 15:12:51 UTC from IEEE Xplore. Restrictions apply.

case happens as often as $1 - \frac{T_{\text{snk}}}{T_{\text{src}}}$, i.e., the synchronization overhead for a signal between these two components can be reduced by the above ratio.

C. Integration with the Platform

To realize the periodic execution of components and thus to ensure their binding to physical time, we require the user to write a simple glue code that we call *task dispatcher*. Its only purpose is to attach to a platform-specific timer that fires every T_b time units, and to delegate this trigger to the individual containers. The actual release schedule (each component C_i is released according to its T_i) is encoded in the containers, invisible to the task dispatcher. Thus, the user does not have to care in which order or even under which policy the components are being scheduled. Moreover, since our library is *thread-safe*, it is possible to make use of multi-threading, to parallelize the computationally intensive step *clock*, such that the execution *physically* takes place in parallel.

VI. Case Study

We used the Pololu 3pi robot as an embedded platform in a bare-metal setup to show the usefulness and simplicity of the proposed approach. The platform is based on an 8-bit Atmel Atmega 328p processor, which has tight constraints on memory consumption and run-time overhead (32 kB flash, 2 kB RAM, 20 MHz).

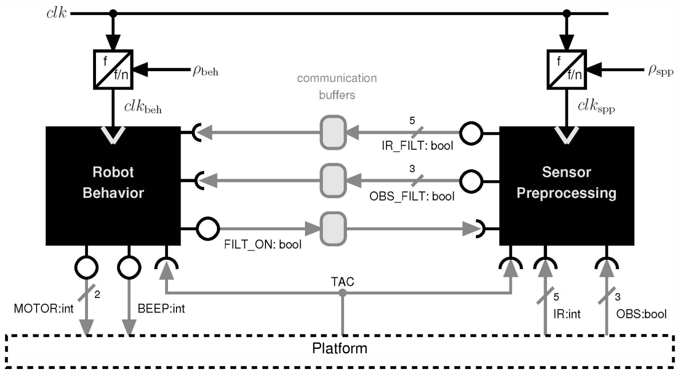

Fig. 4. Setup for the case study: We integrate two black boxes as shown, which together control a robot.

The goal is to let the robot solve a maze drawn with 2 cm-wide lines on the floor. For this we use the *Pledge Algorithm*, which essentially follows the maze's walls until it reaches the exit of the maze. The robot can detect the lines through five reflectance sensors (`IR[0...4]`).

We decided to use two components in this setup: (1) *sensor pre-processing* (de-noise, de-bounce) and (2) *behavioral logic* (the Pledge algorithm). Both are supposed to run in parallel, whereas the sensor pre-processing and the platform I/O run at ten times the frequency of the behavior ($\rho_{spp} = \rho_{io} = 1,\ \rho_{beh} = 10$).

Once the components have been developed independently, the resulting black boxes are encapsulated in two different containers that are connected by signals as depicted in Fig. 4.

A. Deterministic Interface & Portability

The platform's sensors and actuators have to be connected to the components. To avoid non-determinism at this point, we advocate using another component which we call *I/O-Mapper*. It serves as a gateway to the platform functions, i.e., it "offers" all sensors, and "receives" all actuator requests. This way it is ensured that every output has exactly one driver and that sensor signals are consistent among all components.

The code inside the I/O mapper naturally is platform-specific, which is why we implemented this component in C, following the rules in §IIIA. Once the I/O mapper is completed, the user need not be concerned with the platform anymore. As a result, containers running "upon" the I/O mapper can be easily ported to other platforms.

B. Freshness of Sensor Data

The maximum delay of the sensor data can be obtained from the maximum communication delay derived in §IVA. It is produced through the chain of components *I/O Mapper* → *Sensor Pre-processing* → *Behavior*. With T_b = 10 ms, according to Theorem 2, this amounts to 20 ms. Since the robot moves with approximately 12 cm/s, this delay corresponds to a distance of 2.4 mm, which is 12 % of the line width. Consequently the communication delay is sufficiently low to detect a line reliably.

C. Implementation Overhead

We evaluated memory and timing overhead of our library, both for the case study and more generic settings, to show the relationship between used features (components, signals etc.) and resources (memory, processor time).

C.1. Memory Overhead

Methodology: We determined the memory overhead from the output of the `avr-gcc` 4.7.2 compiler/linker, by observing growth in the segments `.text` and `.data`. Compiler optimization was turned off, so as not to distort the results.

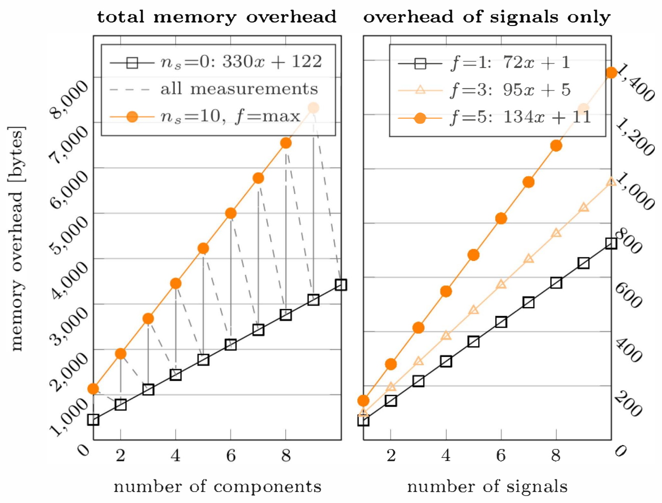

Fig. 5. Memory overhead (Atmel Atmega 328p) of the containers (left) and signals (right).

Case Study: Overall, the program size increased by 8 % from 14,090 to 15,212 Bytes in comparison to a program which just instantiates the components on the robot platform without letting them interact[2]. Consequently this 8 % would be the amount of memory, that would be available for a *manual* composition of the components instead of our library.

[2]Obviously, this implementation was not functional.

Authorized licensed use limited to: UNIVERSITAETSBIBLIOTHEK CHEMNITZ. Downloaded on April 13,2022 at 15:12:51 UTC from IEEE Xplore. Restrictions apply.

Generic: We compiled a large number of different programs and determined only the memory demand incurred by our *containers*, whilst varying the number of outgoing signals per container n_s and also in the number of sinks per signal, i.e., their *fan out*, denoted by f.

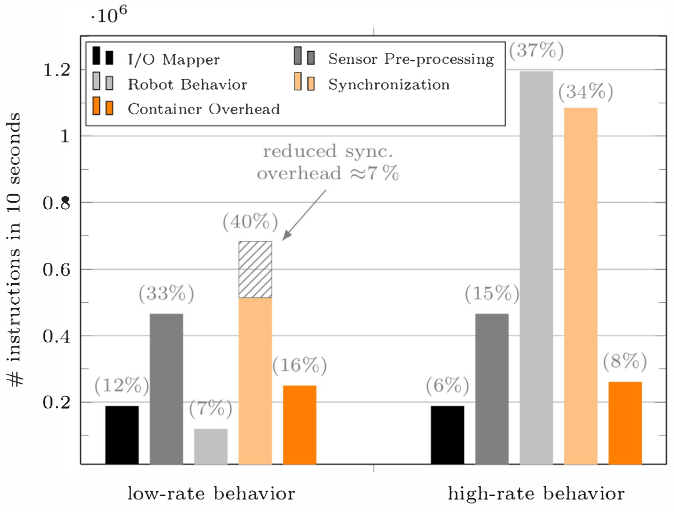

Fig. 6. Timing overhead for the behavioral component running at a low rate (left, $\rho_{\text{beh}} = 10$) and a high rate (right, $\rho_{\text{beh}} = 1$).

The left side of Fig. 5 shows in one plot all the taken measurements (dashed line). It includes two *linear* fits: (1) The first (lower) line captures the memory used by empty components (i.e., n_s=0). It can be seen that each component requires 330 B, and that the basis overhead is 122 B. (2) The other (upper) line goes through all measurements where we used $n_s = 10$ signals, each connected to all components (f=max). The right side of Fig. 5 shows the memory demand per signal, indicating that a higher fan out requires more memory, but also that the sinks share some memory at the signal source, since triplicating fan out results in less than three time the memory.

Summary: Our library has a very low base overhead per component and an additional overhead that scales linearly with the number of components and signals.

C.2. Timing Overhead

Methodology: We determined the overhead in terms of *cycle-based* execution time, to avoid suffering from potential distortions caused by background processes or the measurement itself. We used `callgrind` on again non-optimized code and evaluated only the timing profile of those functions rooting in the task dispatcher.

Case Study: The resulting distribution of execution time for the setup of our case study (prescalers $\rho_{spp} = \rho_{io} = 1$, $\rho_{beh} = 10$) is shown in Fig. 6 at the left-hand side. The plot shows the number of instructions for a simulated time of 10 s, in total 1.71 million. Most of the processing time was spent in synchronization and the sensor pre-processing. The total timing overhead incurred by using our approach is 249,714 instructions, i.e., 16 %.

Generic: We let the behavioral component from the case study run ten times as fast as before, by setting $\rho_{beh} = 1$. The result is shown in Fig. 6 on the right side: As expected, the behavioral component consumes ten times the processing time than before, and both I/O mapper and pre-processing components are unaffected, consuming the same processing time as before. The synchronization, however, increased by (only) a factor of 1.6, as opposed to the 10x rate change of the behavioral component. The total timing overhead is now 261,114 instructions (8 %), i.e., the ***absolute*** timing overhead barely changes. This suggests that our framework comes with a constant timing overhead.

Summary: Our library incurs a timing overhead that only depends on the number of the components. It is independent of the component periods. In contrast to this, the synchronization between components – which is required anyway when integrating components – does depend on periods.

VII. Conclusions

In this paper we proposed a model of computation and a prototypical implementation of a C library supporting the deterministic composition of real-time programs from black-box components. Our library automatically solves the tedious task of implementing communication and synchronization mechanisms between components, and can easily be used on different targets. It is light-weight, easy to use and does not require a special compiler or a runtime environment. The user only has to provide interface specifications, e.g., declare the available and required communication signals. The correct behavior of our library running on a particular target system can be verified using standard methods such as worst-case execution and response time analysis. As future work, we plan to extend our model of communication to allow for shorter communication delays, as well as extending the implementation towards multi-core platforms.

References

[1] Angelov, C., et al. A software framework for hard real-time distributed embedded systems. In *Euromicro Conference SEAA* (Sept. 2008).

[2] Delaval, G., and Rutten, E. A domain-specific language for multitask systems, applying discrete controller synthesis. *EURASIP Journal on Embedded Systems 2007*, 1 (2007).

[3] Edwards, S. A., and Tardieu, O. SHIM: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on VLSI Systems 14*, 8 (2006).

[4] Eidson, J., Lee, E. A., Matic, S., Seshia, S. A., and Zou, J. Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE (special issue on CPS) 100*, 1 (January 2012), 45 – 59.

[5] Feiler, P., Lewis, B. A., and Vestal, S. The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems. In *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications* (Oct 2006), pp. 1206–1211.

[6] Henzinger, T. A., Horowitz, B., and Kirsch, C. M. Giotto: A Time-triggered Language for Embedded Programming. *Proceedings of the IEEE 91*, 1 (2003).

[7] Liu, J., and Lee, E. A. Timed multitasking for real-time embedded software. *IEEE Control Systems 23*, 1 (2003).

[8] Maraninchi, F., and Bouhadiba, T. 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In *ACM International Conference on Generative Programming and Component Engineering (GPCE)* (Oct. 2007).

[9] Scaife, N., and Caspi, P. Integrating model-based design and preemptive scheduling in mixed time-and event-triggered systems. Tech. Rep. TR-2004-12, Verimag, 2004.

[10] Stewart, D. B., et al. Design of dynamically reconfigurable real-time software using Port-Based Objects. *IEEE Transactions on Software Engineering 23*, 12 (1997).

[11] Tignol, J. *Galois' Theory of Algebraic Equations*. World Scientific, 2001.

Authorized licensed use limited to: UNIVERSITAETSBIBLIOTHEK CHEMNITZ. Downloaded on April 13,2022 at 15:12:51 UTC from IEEE Xplore. Restrictions apply.