Programming Support for Time-sensitive Adaptation in Cyberphysical Systems

Mikhail Afanasov  
Politecnico di Milano, Italy  
mikhail.afanasov@polimi.it

Aleksandr Iavorskii  
John Wiley & Sons Inc., Russia  
yavalvas@gmail.com

Luca Mottola  
Politecnico di Milano, Italy and SICS Swedish ICT  
luca.mottola@polimi.it

Abstract—Cyberphysical systems (CPS) integrate embedded sensors, actuators, and computing elements for controlling physical processes. Due to the intimate interactions with the surrounding environment, CPS software must continuously adapt to changing conditions. Enacting adaptation decisions is often subject to strict time requirements to ensure control stability, while CPS software must operate within the tight resource constraints that characterize CPS platforms. Developers are typically left without dedicated programming support to cope with these aspects. This results in either to neglect functional or timing issues that may potentially arise or to invest significant efforts to implement hand-crafted solutions. We provide programming constructs that allow developers to simplify the specification of adaptive processing and to rely on well-defined time semantics. Our evaluation shows that using these constructs simplifies implementations while reducing developers’ effort, at the price of a modest memory and processing overhead.

I. INTRODUCTION

Cyberphysical systems (CPS) enable the tight integration of embedded sensors, actuators, and computing elements into feedback loops for controlling physical processes. Example applications include factory automation, automotive systems, and robotics [25]. CPS operate at the fringe between the cyber domain and the real world [11]. Both the execution of the control logic and the platforms it runs on are thus inherently affected by environmental dynamics [25]. This requires CPS software to continuously adapt to these dynamics. To enact the needed adaptations, developers may employ various approaches, including dynamically changing the control logic.

To complicate matters, control loops are most often timesensitive [24]; the control logic must be executed at a given frequency to ensure the stability of processes. Adaptation is an integral part of the control loop, and thus subject to the same time constraints. Thus, the timing aspects of taking and enforcing adaptation decisions become crucial. Such complex time-sensitive adaptive processing must withstand the strict resource constraints of CPS platforms; the most advanced CPS devices feature 32-bit micro-controller units (MCUs) with tens of KBytes of RAM, while being battery-operated.

As we illustrate in Sec. II, developers are often left without dedicated support to implement adaptive time-sensitive CPS software. This leads to easily overlooking the potential issues related to the timing aspect of run-time adaptation, ultimately affecting the stability of the controlled processes. Fritsch et al. [7] show, for example, that timing aspects are often neglected in developing adaptive automotive software. Similar observations also apply to robot controllers [2], [18]. Whenever developers do recognize these issues, they tend to implement complex hand-crafted solutions, mostly due to the lack of time semantics in mainstream programming abstractions.

To address these issues, we design and implement a custom realization of context-oriented programming [10], [22] that is: i) conceived for resource-constrained embedded devices, and ii) embeds well-specified notions of adaptation modality and time. As described in Sec. III, these notions allow developers to distinguish different ways to schedule an adaptation decision and to place an upper-bound on the time taken to apply such decisions. The former is useful, for example, to avoid functional conflicts when switching from one control logic to another. The latter provides a specified time semantics when adaptation decisions need to abide to real-time deadlines. We render these notions in a dedicated extension of the C++ language we call COP-C++, supported by a corresponding toolchain we develop.

As reported in Sec. IV, we assess our work by quantitatively comparing the complexity of representative implementations of CPS software using traditional programming constructs against those we design. Our results indicate that the developers’ effort is reduced using COP-C++. The cost to gain this benefit is a modest increase in resource consumption, especially in processing time and memory occupation. For example, the worst-case processing overhead we measure through real-world experiments on modern 32-bit MCUs amounts to only 20µs, negligible given the time scales of the considered control loops. We end the paper by discussing related efforts in Sec. V and with brief remarks in Sec. VI.

II. PROBLEM

Consider the need to localize a gas leak in an indoor environment. Tiny aerial drones are envisioned to perform this task efficiently and with minimal cost [4]. Their behavior, as dictated by a given control logic, depends on surrounding conditions, application state, and sensed data [4].

Fig. 1 depicts a possible design for such an application. Initially, every drone moves to a predefined location using
Addressing these issues, however, is also not trivial without proper programming support. For example, issue I3 often emerges as developers intentionally overlap initialization and clean-up operations to increase parallelism when performing I/O operations. Solving issue I1 by simply switching the ordering of clean-up and initialization decreases parallelism, possibly prolonging the time required for switching controllers and thus exacerbating issue I2. To remedy this, developers implement hand-crafted solutions to regulate the time for switching controllers, further impacting the quality of implementations, making issue I1 even worse.

III. COP-C++

Context-oriented programming (COP) [10] is a paradigm to simplify the implementation of adaptive software. It fosters a strict separation of concerns, achieved through two key notions: i) the different situations where the software needs to operate are mapped to different contexts, and ii) the context-dependent behaviors are encapsulated in layercd functions, that is, functions whose behavior changes—transparently to the caller—depending on the active context.

COP already proved effective in creating adaptive software in mainstream applications, such as user interfaces [13] and text editors [12]. To that end, COP extensions of popular high-level languages, such as Java and Erlang, emerged [12], [22]. Embedding COP extensions within an existing language, however, often relies on features such as reflection and dynamic binary loading, which are difficult to implement on resource-constrained platforms, such as those employed in CPS.

A. Context-constrained C++

To address issue I1 in Sec. II, we embed the key COP abstractions within C++, as it arguably represents a large fraction of the existing codebase in CPS. The resulting language, called COP-C++, retains the original C++ semantics with the addition of custom semantics and keywords. We focus on the local adaptation, while any distributed functionality is orthogonal to our efforts. Indeed, our approach can be used with any middleware for adaptation in distributed systems; for example, i.Land [8].

For simplicity, we illustrate the language through examples here. The full grammar is publicly available together with the corresponding tool-chain.1

Context groups and individual contexts. Similar to previous work [1], we group together contexts sharing common characteristics, such as the functionality provided or the environment dimension that drives the transition between contexts. In the example of Sec. II, individual contexts would map to the corresponding tool-chain.1

---

1. https://bitbucket.org/neslabpolimi/cop_c++_translator
context group FlightControlGroup {
  context Hovering; context Navigate; context LeakLoc;
  context Landing;
  public:
  1. layered void step() = 0;
}

Fig. 3: Example definition of context group.

class LeakLoc : private FlightControlGroup {
  public:
  1. LeakLoc() : _ticker(new Ticker()) {};
  2. virtual 'LeakLoc(){};
  3. private:
  4. layered void step() { /* ... controller functionality */}
  5. bool initialize() {_ticker->attach(broadcast, 0.33)};
  6. void cleanup() {_ticker->detach()};
  7. void broadcast() { /* broadcast routine */
    _ticker->_tickers();
  }
}

Fig. 4: Example implementation of individual context.

Fig. 5: Lazy and fast activation of a new context.

Comparing to plain C/C++, as shown in Fig. 2, the code appears much simplified. No global variables are necessary to keep track of the current controller. No cumbersome switch statements are required, either. Only a single call to the step function appears in the code, which is automatically dispatched to the active context. The interleaving of controllers’ initialization and clean-up while switching, as well as the corresponding time semantics, are completely encapsulated in the aforementioned qualifiers, as explained next.

B. Qualifiers

As described in Sec. III, functionality meant to operate in different situations may conflict with each other during the switch—as per issue I2. In addition, potential issues may emerge as current implementations enforce no time limit on the execution of adaptation decisions, as per I3. To help programmers cope with these issues, the qualifiers lazy and fast[within] indicate different modes to enact adaptation decisions and possibly specify time constraints.

Modes. Fig. 5 illustrates the difference in performing a context switch using lazy or fast. Using lazy, the clean-up of the previous context needs to complete before initializing the new context. As a result, no functional conflicts may ever arise. However, during the switch, the system rests in an uncertainty state where no context is active. A call to a layered function while switching, the system may be in a state of uncertainty, where no context is active. A call to a layered function within this time results in no operation. In addition, as no parallel execution occurs, the latency grows as the sum of the time to clean-up from the previous context and the time to initialize the new one.

In contrast, as shown in Fig. 5, using fast the system first initializes the new context; then performs the clean-up of the previous one. If some operations inside initialize are non-blocking and may be asynchronously executed, such as those involving I/O operations, fast allows the system to increase parallelism. As a result, the time to apply an adaptation decision reduces, yet programmers must take additional care to avoid functional conflicts between contexts during the switch. There is indeed a time where both contexts might be possibly simultaneously executing.

Deadlines and rapid context switches. To control the time invested in switching between contexts, programmers may define an optional activation deadline. Say, for example, that in the scenario of Sec. II the context switch needs to complete within $T$ ms to let the new controller perform the actuation within the next $(10 - T)$ ms. The optional qualifier within allows programmers to specify the upper-bound $T$ on the time to switch contexts, as shown in Sec. III-A. Should the upper bound be violated, the initialization is interrupted and the programmer is notified through a callback, which can be used to implement application-specific countermeasures.

All the activate commands are placed in a queue that is asynchronously checked by the system. The latter pulls out the
first **activate** command and executes it. In an emergency situation, such as a collision threat, programmers may want to switch the context immediately. To this end, an instruction **activate FlightControlGroup::Landing immediately** cancels all pending **activate** commands and immediately switches to the **Landing** context.

The qualifiers we discuss here also naturally apply across multiple context groups, defined as explained in Sec. III-A, in applications with parallel adaptive controllers.

### C. Translator

We implement the translator as an extension to the CDT plug-in for Eclipse. It allows programmers to translate from COP-C++ to pure C++ and to rely on standard toolchains to generate executable binaries. First, the translator ensures the consistency of the context-oriented design. For example, a context may not implement any layered functions or not belong to any context group. In this case the translation stops and the developer is informed. Second, the generated source code remains human readable; programmers can modify it to perform further fine-grained optimizations and tuning.

### IV. Evaluation

We assess the effectiveness of COP-C++ along several dimensions. Sec.IV-A quantitatively demonstrates the benefits of COP-C++ in complexity of the implementations and required programming effort. Such benefits come at a price of processing and memory overhead, which we report in Sec.IV-B. We compare the performance of different combinations of qualifiers for context switch in Sec.IV-C, whereas Sec.IV-D shows the programming effort required to manually cope with functional conflicts when using **fast** switching.

Our evaluation targets modern 32-bit ARM Cortex M MCUs that are often employed in CPS. We employ STM Nucleo prototyping boards equipped with Cortex M3 MCUs running at 32 MHz and 80 KBytes of RAM. The architecture of these is similar, if not the same, to devices employed in real-world CPS applications, while the board also offers convenient testing facilities that enable the kind of fine-grained measurements we discuss next.

As input to our evaluation, we implement the gas leak localization application described in Sec.II using COP-C++. We use two functionally-equivalent implementations as baselines. The first one uses pure C++ by following the structure of an original ArduPilot-like implementation, discussed in Sec.II. We call this implementation **PUREC++**. The second baseline is similar to **PUREC++**, with the addition of manually-implemented functionality to control the time for switching controllers, that is, the same functionality the **within** qualifier provides declaratively. Such a baseline is instrumental to examine the difference between the manual implementation and the automatic generation of this functionality. We call this implementation **TIMEC++**. We implement all three versions of the application using the mBed [15] libraries provided by STM. We use the standard ARM gcc tool-chain for compiling.

#### A. Complexity and Effort

We compare the complexity and efforts for the three implementations using Halstead metrics [9]. These are intended to investigate the properties of source code independently of the programming language. Halstead et al. use four basic metrics: the total number of operands (OP), the total number of operators (OD), the number of unique operators (UOP), and the number of unique operators (UOD). An operator is a language-specific keyword, whereas variables and constants are operands. For example, in Fig.4, **layered** is an operator, whereas **ticker** is an operand. Based on the four basic metrics, other metrics are derived as shown in Fig.6.

#### Results

Fig.7 reports the values of the Halstead metrics for the three aforementioned implementations. The **Difficulty** in a COP-C++ implementation is slightly higher than for the baselines. This is due to the additional keywords we add to define context groups and individual contexts, as well as the use of qualifiers.

On the other hand, COP-C++ reduces the **Volume** of the program; therefore, maintenance and debugging should be facilitated as programmers need to absorb less information to understand a program. Programmers also spend less **Effort** to realize the program in the first place. The benefits of COP-C++ in these regards become even more apparent when considering **TIMEC++**. In this case, both the **Volume** and **Effort** further increase, amplifying the benefits of COP-C++.

#### B. Processing and Memory Overhead

The benefits above incur a run-time cost in processing time and memory occupation. To assess these, we separately compare a COP-C++ implementation that only uses the **fast** qualifier against **PUREC++**, and a COP-C++ implementation that also employs the **within** qualifier against **TIMEC++**. The original Ardupilot implementation only uses a kind of controller switch similar to the semantics of the **fast** qualifier, so we do not study here the run-time overhead for the **lazy** one. We investigate this in Sec.IV-C.

According to Fig.1, different environmental events may trigger the adaptation. We emulate these events on the Nucleo board as external interrupts through GPIO pins. We use a Tektronix TBS 1072B-EDU oscilloscope attached to the board.
for the former, the latency grows by only marginally increases the latency in switching context. As this aspect in Sec. IV-D.

Results. Fig. 8 shows the processing time to switch controller depending on the external event. Adaptation in COP-C++ takes slightly more time—approximately $11\mu Sec$—compared to PUREC++. The absolute values vary due to different initialization routines; for example, periodic beaconing is only initialized when LeakLocalization is activated by alert beacon or high gas concentration. With the within qualifier, the processing overhead compared to TIMEC++ reaches $20\mu Sec$. Such a penalty, however, is still almost unnoticeable, as the typical control loop runs at hundreds of Hz.

COP-C++ shows a mere 200B RAM overhead, which is negligible compared to both baselines that consume 2.1kB of RAM. RAM consumption is often an issue when developing CPS software; therefore, minimizing the impact on this figure is key. On the other hand, the program memory usage using COP-C++ appears 14.6kB higher than in the baselines that use 20kB. This is mainly due to the simplified implementation of the control loops we employ for this study, where only basic functionality are included and platform-specific libraries are replaced with empty stubs. Functionally-complete implementations are much larger; for example, the full ArduPilot [2] requires 792 KB of program memory. A major fraction of these are not processed by our translator; therefore, we expect the relative overhead in program memory to amortize.

C. Qualifiers

As the example application in Sec. II only uses the fast qualifier, we quantitatively study here the trade-offs between the adaptation qualifiers in COP-C++, described in Sec. III-B. The memory overhead is the same independent of the specific combination of qualifiers that appear in the code, and corresponds to the values shown in Sec. IV-B. Therefore, here we focus on the latency to perform the context switch depending on the combination of qualifiers. The experimental setup is the same as in Sec. IV-B.

Results. Our investigations show that the lazy qualifier requires $33, 6\mu Sec$ to complete the context switch. This latency is the price programmers pay to ensure that no functional conflicts arise during the switch. To reduce this time, programmers can use the fast activation type that requires only $21, 4\mu Sec$ without optional qualifier within. This choice, however, requires programmers to handle potential functional conflicts by hand, increasing the programming effort. We investigate this aspect in Sec. IV-D.

Using the optional within or immediately qualifier only marginally increases the latency in switching context. As for the former, the latency grows by $10\mu Sec$ because of the additional processing required to initialize a dedicated timer that fires if the switch takes too much time. In the latter case, the additional processing time amounts to only $\approx 1\mu Sec$, required to purge the queue of pending context switches. In both cases, the absolute values are very limited, and they should not impact the timings of control loops that typically run with periods that are orders of magnitude larger.

D. Development Trade-offs

Using the fast qualifier may result in functional conflicts because the initialization and clean-up routines of different contexts overlap in time. Programmers need to handle these conflicts by hand. To assess the additional programming effort required, we implement a new version of the application in Sec. II with the addition of simple wrappers around any of the classes where asynchronous events may fire during a fast context switch. These include timers and classes that signal hardware interrupts. The wrappers intercept any such asynchronous event and forward it further only after checking that the active context corresponds to the one the event is addressed to. Cleaner, yet more complex solutions are also possible. Considering simple wrappers provides a lower-bound in terms of the additional programming effort.

We assess the added programming effort by re-calculating the Halstead metrics of Sec. IV-A on the new implementation. Further, the wrappers obviously add latency to the context switch. We measure this with the same setup of Sec. IV-B.

Results. As shown in Fig. 9, the wrappers add a mere $\approx 2\mu Sec$ in latency during a context switch. Thus, their performance impact is negligible. However, the complexity of the implementation increases considerably, as reported in Fig. 10. Programmers need to invest significant efforts not only in implementing the wrappers, but also to nail down all the classes that could possibly lead to functional conflicts, and provide a wrapper for each of these. Fig. 10 reports a considerable increase in all the complexity metrics when wrappers are used. For example, the Volume of the source code increases by 80%, the source code is 53% more Difficult, and requires almost 3 times the Effort to be written.

V. RELATED WORK

Time-sensitive software adaptation in CPS is a multi-facted problem. Albeit comprehensive programming support largely
lacks, works exist that tackle individual aspects.

**Parameter and component configurations.** Adjusting the software operating parameters is one way to adapt. For example, Garcia-Valls et al. [8] focus on the reconfiguration of the nodes in distributed soft real-time system that meets stated performance goals. In the area of adaptive controllers, Mokhtari et al. [17] and Frew et al. [6] tune the operation of unmanned aerial vehicles (UAVs) based on sensor inputs. These approaches focus on adapting a specific fraction of the system’s functionality, for example, motors’ parameters or nodes’ configuration, and cannot be applied where the whole control logic must be changed on a single node. Our work does not focus on the mechanism to adapt a specific functionality, but provides generic programming support for implementing adaptive CPS software under time constraints.

In component-based systems, software reconfiguration often occurs by plugging components in/out or by changing component wirings [21]. Dedicated component models exist that allow developers to verify—using formal techniques such as model-checking—the correctness of new component configurations [20]. Some of these works focus on specific application domains such as automotive [26] and autonomous underwater vehicles [16]. Unlike our work, these approaches offer no programming support to deal with enacting time-sensitive adaptation decisions.

**Programming support for adaptation.** Software adaptation for traditional platforms is extensively studied. Some of the works explicitly focus on programming support. For example, COP [10] itself was implemented in a number of high-level languages [3], [12], [22], [23]. The techniques normally used to embed COP abstractions into a host language tend to be impractical in CPS because of resource constraints. Similar observations apply to Meta- and Aspect-oriented programming (AOP) [22]. The corresponding abstractions often require self-modification of the deployed binaries [14], which is hard to implement on resource-constrained platforms. Our work renders COP concepts amenable for implementation on typical CPS devices, while adding semantics useful when enacting time-constrained adaptation decisions.

The need to provision programming support for time-sensitive adaptation was also recognized in the area or real-time operating systems (RTOSes). Dedicated programming abstractions based on reflection were added to existing RTOSes [19] or specific services were made available that perform the needed reconfiguration in a safe manner [5]. These attempts utilize language- or operating system-specific features that are often not available in typical CPS platforms, because of resource-constraints. Target platforms of these approaches are either the traditional computing machines [8], [21] or FPGAs [5], which greatly surpass CPS platforms in terms of available resources and energy consumption. Our solution, instead, is designed for resource-constrained devices, does not require any language-specific features such as reflection, and remains decoupled from the the underlying operating system.

VI. Conclusion

We presented COP-C++, an extension to C++ we expressly conceived to simplify the implementation of time-sensitive CPS software. To that end, we borrowed concepts from COP and realized them in a way that is feasible on resource-constrained devices, while adding semantics to govern the time aspects during the adaptation process. We implemented a dedicated translator from COP-C++ to pure C++. Our quantitative evaluation shows that COP-C++ simplifies implementations of paradigmatic CPS functionality while reducing programmers’ effort, at the price of a modest run-time processing and memory overhead. For example, processing overhead in our experiments is limited to tens of µSec, while RAM overhead is negligible. Program memory overhead, on the other hand, should be amortized with the increasing size of implementations.

References