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.

1. 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.

Control loops are most often time-sensitive [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. 2, 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, 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. 3, 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 tool-chain we develop.

As reported in Sec. 4, 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 con-

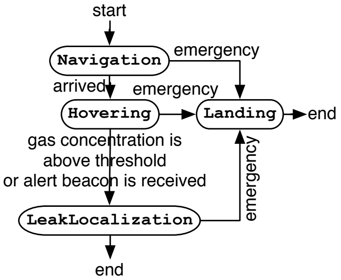

Figure 1: Software controller for gas leak localization.

```
1  static int8_t current_controller = NONE;
❷ static void step(){// is called at 100+Hz
3    switch (current_controller) {
4      case NAVIGATE: navigate_step(); break;
5      case HOVERING: hovering_step(); break;
6      case LEAK_LOC: leak_loc_step(); break;
7      case LANDING: landing_step(); break;}}
❽ static bool set_controller(uint8_t controller) {
9    if(controller==current_controller){return true;}
10   bool success = false;
11   switch(controller) {
12     case HOVERING: success=hovering_init(); break;
13     case LEAK_LOC: success=leak_loc_init(); break;
14     case NAVIGATE: success=navigate_init(); break;
15     case LANDING: success = landing_init(); break;
16     default: success = false; break;}
17   if (success) {// update controller
⓲     exit_mode(current_controller, controller);
⓳     current_controller = controller;
20   } else {// log error
21     Log_Write_Error(controller);}
22   return success;}
```

Figure 2: Example implementation of adaptive controller.

trol loops. We end the paper by discussing related efforts in Sec. 5 and with brief remarks in Sec. 6.

2. 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 a Navigation controller. Upon arriving, a drone uses a Hovering controller to sample the gas concentration. Whenever it detects a value above a threshold, a drone switches to a LeakLocalization controller that broadcasts alert beacons via a low-range radio. Nearby drones that receive the beacon also switch to the LeakLocalization controller, and come closer to the drone that initially detected the leak to obtain finer-grained measurements. In case of emergencies, such as a hardware failure, the Landing controller is activated.

Fig. 2 depicts an example implementation of the required adaptive behavior. The structure of the code reflects real implementations in the considered systems; for example, in the Ardupilot [2] autopilot for drones. Control is triggered in function step() in line ❷, which is called at 100Hz. Depending on the global variable current_controller, different concrete controllers are executed. Controller adaptation is implemented in function set_controller() in line ❽. Depending on what controller is to be activated, an individual controller is first initialized, the clean-up routine of the previous controller is executed in line ⓲, and the global variable indicating the active controller is updated in line ⓳.

Despite being simplified, the code in Fig. 2 already shows several issues, some not even entirely evident:

I1) The processing is strictly *coupled* with the different controllers. For example, adding a new controller, or removing an existing one would require changing the code in several places. On the other hand, resource constraints prevent using high-level languages that ameliorate these issues. As a result, implementations are typically entangled and thus difficult to debug and to maintain.

I2) In Ardupilot, control runs at 100 Hz: every 10 *ms* a controller must perform the necessary actuation. However, current implementations enforce *no time limit* on adaptation. This may become an issue when executing set_controller: should a controller's initialization or clean-up take too long, the controller will not be executed in time, which may affect the system's stability.

I3) When switching controller, the previous and new controllers may *conflict* with each other. For example, the LeakLocalization controller includes a periodic task to transmit alert beacons that may still operate when the Landing mode is possibly initialized. This would mean the drone keeps beaconing also when it lands, which may affect the system's correctness. Asynchronous operations, such as interrupt handlers firing while switching controller, may create similar issues.

As we argued earlier, these problems are often overlooked by CPS developers. Issue I1 impacts the quality of implementations, whereas issue I2 and I3 potentially affect dependability. 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.

3. COP-C++

Context-oriented programming (COP) [10] is a paradigm to simplify the implementation of adaptive software. It is based on 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 *layered* functions, that is, functions whose behavior changes—transparently to the caller—depending on the active context.

COP 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.

3.1 Context-oriented C++

To address issue I1 in Sec. 2, 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, iLand [8].

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

[1] https://bitbucket.org/neslabpolimi/cop_cpp_translator

```
❶ context group FlightControlGroup {
❷  context Hovering;context Navigate;context LeakLoc;
❸  context Landing;
 4  public:
❺    layered void step() = 0;};
```

Figure 3: Example definition of context group.

```
❶ context LeakLoc : private FlightControlGroup {
 2   public:
 3     LeakLoc(): _t(new Ticker()) {};
 4     virtual ~LeakLoc();
 5   private:
❻     layered void step(){//controller functionality}
❼     bool initialize(){_t->attach(&broadcast,0.3);}
❽     void cleanup() {_t->detach();}
 9     void broadcast(){// broadcast routine}
❿     Ticker* _t;};
```

Figure 4: Example implementation of individual context.

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. 2, individual contexts map to the individual controllers in Fig. 1. These contexts would be grouped together as they all provide control functionality.

Fig. 3 shows the specification of a context group in the application of Sec. 2 using COP-C++, which extends the notion and syntax of C++ `class`es. Context groups are declared with the keyword `context group`, as shown in line ❶. Inside a context group, programmers declare the list of contexts that belong to the group, as in lines ❷ to ❸. In addition, they specify the layered functions that the group offers to other classes. These are indicated using the `layered` keyword, as in line ❺, and implemented differently by the individual contexts.

Only one context is active inside a group at every point in time; the active context is in charge of executing the corresponding layered functions. As a result, the context group acts as a container that hides the individual contexts from the users of a specific functionality. This helps developers decouple context-dependent functionality from their use. Further, it makes it possible to statically generate the code to dispatch layered function calls to the corresponding implementation in the active context, ameliorating the need for advanced language features hardly feasible on resource-constrained platforms.

In COP-C++, individual contexts also extend the notion and syntax of C++ `class`es. Fig. 4 shows an example based on the application of Sec. 2. The `context` keyword in line ❶ specifies that this class is an individual context, part of the `FlightControlGroup` it inherits from. In line ❻, the programmer implements the context-dependent behavior for the layered function `step` defined in the corresponding context group. In every individual context, programmers may add initialization and clean-up routines with the predefined `initialize()` and `cleanup()` methods, as in lines ❼ and ❽. These are useful when starting a new controller or stopping the currently executing one.

Adaptation. Using COP-C++, enacting an adaptation decision that prompts to easily switch between the controllers. For example, the command `activate FlightControlGroup::Hovering fast within 5ms` performs the change of the currently-executing controller including initialization and clean-up. The specific scheduling of these operations depends on the qualifiers `lazy` and `fast[within]` in the same lines, whose semantics we explain next. The call to function `FlightControlGroup::step()` is then transparently dispatched to the currently active context.

Compared to plain C/C++, as shown in Fig. 2, the code is 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.

3.2 Qualifiers

As described in Sec. 2, 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 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.

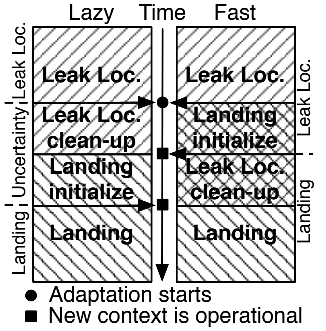

Figure 5: *Lazy* and *fast* activation of a new context.

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. 2 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. 3.1. 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. 3.1, in applications with parallel adaptive controllers.

3.3 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 further optimize it and tune.

4. EVALUATION

We assess the effectiveness of COP-C++ along several dimensions. Sec. 4.1 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. 4.2. We compare the performance of different combinations of qualifiers for context switch in Sec. 4.3, whereas Sec. 4.4 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. 2 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. 2. 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` qualifiers 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.

4.1 Complexity and Effort

We compare the complexity and efforts for the 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 operands (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 `_t` is an operand. Based on the four basic metrics, other metrics are derived as in Tab. 1.

Table 1: Definition of Halstead metrics.

Name/Definition	Description
Length (LTH) $OP + OD$	Shows how long is a program in terms of "words", where a "word" is an operator or an operand.
Vocabulary (VOC) $UOP + UOD$	Indicates the number of unique "words" in a program.
Difficulty (DIF) $UOP/2 \times OD/UOD$	Indicates how difficult it is to understand the program.
Volume (VOL) $LTH * \log_2 VOC$	Refers to how much information does a reader have to absorb to understand the program semantics.
Effort (EFF) $DIF * VOL$	Reflects the effort required to recreate, or to initially write the program.

Table 2: Halstead metrics applied to the COP-C++ implementation of the application in Sec. 2 against baselines.

Metric	pureC++	timeC++	COP-C++
Operators count (OP)	981	1068	750
Unique operators (UOP)	32	33	36
Operands count (OD)	439	478	383
Distinct operands (UOD)	121	124	101
Program length (LTH)	1420	1546	1133
Program vocabulary (VOC)	153	157	137
Difficulty (DIF)	58	61	68
Volume (VOL)	10305.49	11227.48	8042.07
Effort (EFF)	597718.46	687926.5	546860.78

Results. Tab. 2 reports the values of the Halstead metrics for the 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 simpler as programmers need to absorb less information to understand a program. Programmers spend less *Effort* to realize the program in the first place. The benefits of COP-C++ in these regards become even more evident when considering timeC++. In this case, both the *Volume* and *Effort* further increase, amplifying the benefits of COP-C++.

4.2 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. 4.3.

According to Fig. 1, different environmental events may trigger the adaptation. We emulate them on the Nucleo board as external interrupts through GPIO pins. We use a Tektronix TBS 1072B-EDU oscilloscope attached to the board to measure the controller switching time. Memory usage statistics are obtained from the mBed [15] on-line IDE.

Results. Fig. 6 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 acti-

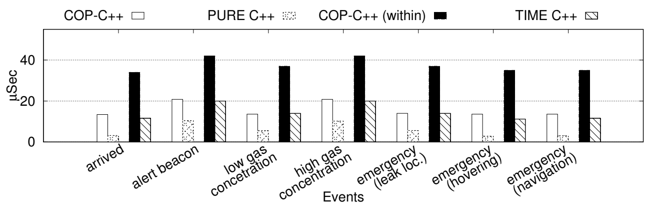

Figure 6: Processing time for switching controller depending on the external event.

vated by *alert beacon* or *high gas concentration.* With the `within` qualifier, the processing overhead compared to TIMEC++ reaches $20\mu Sec$. Such a penalty 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.

4.3 Qualifiers

As the example application in Sec. 2 only uses the `fast` qualifier, we quantitatively study here the trade-offs between the adaptation qualifiers in COP-C++, described in Sec. 3.2. 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. 4.2. 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. 4.2.

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. 4.4.

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.

4.4 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. 2 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.

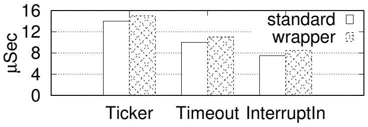

Figure 7: Time to handle asynchronous events, either using standard handlers or wrappers.

Table 3: Halstead metrics for the application of Sec. 2 using COP-C++ with and without manually-implemented wrappers to cope with functional conflicts during a context switch.

Parameter	COP-C++	COP-C++ with wrappers
Operators count	750	1237
Distinct operators	36	52
Operands count	383	646
Distinct operands	101	160
Program length	1133	1883
Program vocabulary	137	212
Difficulty	68	104
Volume	8042.07	14551.67
Effort	546860.78	1513374.12

We assess the added programming effort by re-calculating the Halstead metrics of Sec. 4.1 on the new implementation. Further, the wrappers obviously add latency to the context switch. We measure this with the same setup of Sec. 4.2.

Result. As shown in Fig. 7, 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 Tab. 3. 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. Tab. 3 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.

5. 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 nodes' reconfiguration 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 mainstream 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.

6. 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.

7. REFERENCES

[1] M. Afanasov et al. Context-oriented programming for adaptive wireless sensor network software. In *Proc. of DCOSS*, 2014.

[2] ArduPilot. www.ardupilot.com.

[3] J. E. Bardram. The Java context awareness framework (JCAF) – A service infrastructure and programming framework for context-aware applications. In *Proc. of PERVASIVE*, 2005.

[4] T. R. Bretschneider and K. Shetti. Uav-based gas pipeline leak detection. In *Proc. of ARCS*, 2015.

[5] Y. Eustache and J. P. Diguet. Reconfiguration management in the context of rtos-based HW/SW embedded systems. *J. Emb. Sys.*, 2008.

[6] E. W. Frew et al. Adaptive receding horizon control for vision-based navigation of small unmanned aircraft. In *Proc. of ACC*, 2006.

[7] S. Fritsch et al. Time-bounded adaptation for automotive system software. In *Proc. of ICSE*, 2008.

[8] M. Garcia Valls et al. iland: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems. *IEEE Transactions on Industrial Informatics*, 9(1):228–236, 2013.

[9] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. 1977.

[10] R. Hirschfeld et al. Context-oriented programming. *Journal of Object Technology*, 2008.

[11] M. Jackson. The world and the machine. In *Proc. of ICSE*, 1995.

[12] T. Kamina et al. EventCJ: A context-oriented programming language with declarative event-based context transition. In *Proc. of AOSD*, 2011.

[13] R. Keays et al. Context-oriented programming. In *Proc. of MobiDe*, 2003.

[14] G. Kiczales et al. Aspect-oriented programming. In *Proc. of ECOOP*, 1997.

[15] mBed on-line IDE. developer.mbed.org.

[16] C. McGann et al. Adaptive control for autonomous underwater vehicles. In *Proc. of the AAAI*, 2008.

[17] A. Mokhtari and A. Benallegue. Dynamic feedback controller of euler angles and wind parameters estimation for a quadrotor unmanned aerial vehicle. In *Proc. of ICRA*, 2004.

[18] OpenROV. www.openrov.com.

[19] A. Patil and N. Audsley. An application adaptive generic module-based reflective framework for real-time operating systems. In *Proc. of RTSS*, 2004.

[20] F. J. Rammig et al. Designing self-adaptive embedded real-time software – towards system engineering of self-adaptation. In *Proc. of SBESC*, 2014.

[21] J. C. Romero and M. Garcia-Valls. Scheduling component replacement for timely execution in dynamic systems. *Software: Practice and Experience*, 44(8):889–910, 2014.

[22] G. Salvaneschi et al. Context-oriented programming: A software engineering perspective. *J. Syst. Softw.*, 2012.

[23] S. Sehic et al. COPAL-ML: A macro language for rapid development of context-aware applications in wireless sensor networks. In *Proc. of SESENA*, 2011.

[24] J. A. Stankovic et al. Real-time communication and coordination in embedded sensor networks. *Proceedings of the IEEE*, 91(7), 2003.

[25] J. A. Stankovic et al. Opportunities and obligations for physical computing systems. *IEEE Computer*, 38(11), 2005.

[26] M. Trapp et al. Runtime adaptation in safety-critical automotive systems. In *Proc. of SE*, 2007.