Poster: Programming Support for Time-sensitive Software Adaptation in Cyberphysical Systems

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

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

Carlo Ghezzi
Politecnico di Milano, Italy
carlo.ghezzi@polimi.it

Abstract

We present programming constructs that allow developers to gain control on the timing aspect when enforcing adaptation decisions in Cyberphysical System (CPS) software. The operation of CPS often depends on environmental conditions; as a result, the software must adapt to the changes in these conditions. Often, CPS software is also time-sensitive; for example, when implementing control loops. As a result, the timing of enforcing adaptation decisions becomes crucial. However, developers are left without dedicated programming support to cope with these aspects, which leads either to simply neglect them or to invest additional effort to realize hand-crafted solutions. The programming constructs we conceive allow developers to rely on well-specified semantics when triggering adaptations, and to define time boundaries that the adaptation process must adhere to. We argue that this greatly simplifies the implementation of time-sensitive adaptive CPS software, at the price of a very modest run-time overhead.

1 Overview

Many CPSs implement some form of control loop to take actions on the environment based on sensor inputs [6]. Such control loops are often time-sensitive [5]. By the same token, the operation of such control loops is frequently depending on environment dynamics. To deal with such dynamics, CPS software may implement various forms of adaptation, including dynamically changing the control logic itself.

Problem. For example, our investigations on the control software for autonomous mobile sensing vehicles [1, 3, 4] show that little to no programming support is offered to CPS developers to deal with time-sensitive adaptation of control loops. As a result, developers may simply overlook the potential issues arising in the run-time change of control logic, which affects the system's dependability. Otherwise, developers manually implement ad-hoc functionality to deal with these issues. This may result in significant additional effort, as it is currently a one-off activity.

International Conference on Embedded Wireless Systems and Networks (EWSN) 2016
15–17 February, Graz, Austria

ISBN: 978-0-9949886-0-7

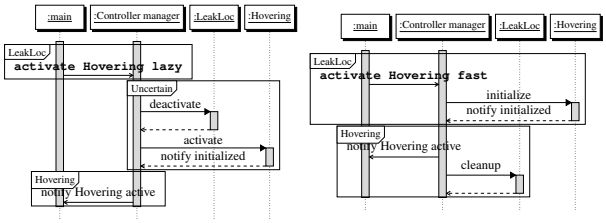
```
1  static bool set_control_loop(uint8_t controller) {
2    bool success = false;
3    switch(controller) {
4      case NAVIGATION:
❺        success = navigation_init();break;
6      case HOVER:
❼        success = hover_init();break;
8      case LEAK_LOC:
❾        success = leak_loc_init();break;
10   // ...
11   if (success) {
⓬     exit_mode(current_controller, controller);}
13   return success;}
```

Figure 1. Example implementation of adaptation routine, borrowed from ArduPilot [1].

Consider an application to localize gas leaks in an indoor environment using aerial drones equipped with gas sensors [2]. Initially, every drone moves to a predefined location using a **`Navigation`** controller. Upon arriving, drones switch to a **`Hovering`** controller to sample the gas concentration. Whenever a drone detects a high gas concentration, it switches to a **`LeakLocalization`** controller that disseminates alert beacons in the vicinity, using a low-range radio. All other drones that receive this beacon also switch to the **`LeakLocalization`** controller to come closer to the one that initially detected the leak. This allows the drones to obtain more fine-grained measurements of the relevant area.

Figure 1 depicts an example implementation of the required dynamic change of controller logic. The structure of the code in the example reflects real implementations; for example, the ArduPilot codebase [1]. The **`set_flight_mode()`** function is called whenever a switch in the controller logic is needed; for example, in line ❺, ❼, and ❾. The function initializes the required controller and performs necessary clean-up of the previous controller, as shown in line ⓬. Such a seemingly simple example conceals two issues:

1) As it stands, the code initializes the next controller first, and then performs the clean-up of the previous one. In other words, there is a time when both controllers are active simultaneously. For example, there might be periodic tasks launched from within the previous controller, such as the periodic beaconing of **`LeakLocalization`**,



(a). Lazy activation. (b). Fast activation.

Figure 2. Activation types.

whose scheduling happens after the next controller is already active. The system's behavior in such a situation is hard to foresee; it is generally unclear how the underlying run-time support takes care of these potential conflicts.

2) Independent of the first issue, the time allowed for switching controller may be upper-bound. On aerial drones, for example, the control logic runs as fast as 100 Hz. This means that a complete change of control logic needs to happen within a 10 ms bound, or the drone may be left with no active controller for at least a full control cycle. Handling these situations is typically left to the programmers' ingenuity and skills, provided the problem is understood in the first place.

Contribution. We address these issues with dedicated programming concepts and corresponding language constructs:

1) To gain control on the interleavings between controllers to be activated and deactivated, we define two *activation types* with specified semantics. The concept allows programmers to trade the latency in switching controller against the risk of potentially harmful conflicts whenever multiple controllers are simultaneously active.
2) We define a notion of *activation deadline* that programmers use to specify an upper bound on the controller switching time. Whenever the deadline is violated, programmers are explicitly notified so they can apply proper countermeasures, while the previous controller is automatically re-instantiated.

We argue that these simple concepts ease the programming burden in implementing time-sensitive adaptation. We materialize these concepts as described next.

2 State of Play

We currently target C++ implementations running atop ARM Cortex M microcontrollers, which are arguably representative of a vast class of modern CPS—including the aerial drone platforms we mentioned earlier.

Activation types. We define two such types, which cover opposite extremes of the same spectrum.

In *lazy* activation, shown in Figure 2a, the clean-up routines of the controller to be deactivated are ensured to complete before any initialization routine of the controller to be activated ever starts. This processing is encapsulated in a single C++ instruction **`activate <CONTROLLER> lazy`**, where **`CONTROLLER`** is the name of a C++ class implementing a default interface. This single instruction can be placed anywhere in the code and replaces the processing of Figure 1 while ensuring the stated semantics, guaranteeing that no conflicts between simultaneously active controllers occurs. However, Figure 2a shows that the system rests in an *uncertain* state where no controller is active, while the latency to switch controller grows as the sum of the time to clean-up from the previous controller and to initialize the new one.

In contrast, Figure 2b shows a case of *fast* activation, where the semantics aims to reduce the latency for the new controller to become active. In fact, programmers are notified of the new controller completing the initialization before starting the clean-up from the previous controller. This may be advantageous, for example, when either routine should wait on I/O operations. The processing is triggered using a C++ instruction **`activate <CONTROLLER> fast`**. As mentioned before, this scheduling of operations may lead to functional conflicts, which generally depend on the involved controllers. A general solution is thus hard to conceive. The straightforward approach is, for example, to wrap the individual controllers to ensure that their functionality does not affect the system during the switch.

Activation deadlines. Programmers may append a modifier **`within <T>`** after the activation instructions to specify an upper bound of **`T`** time units on the controller switch. For example, should the switch to the **`Hovering`** controller happen using **`lazy`** activation within 5 ms, programmers specify **`activate Hovering lazy within 5 ms`**. If the upper bound is violated, the switch of controller stops and the previous controller is re-instantiated. This processing happens transparently to programmers, who are asynchronously notified of the issue and can possibly react with application-specific counteractions.

Feasibility. We check the feasibility of the semantics described above on our target platforms, specifically using an STM Nucleo STM32L152 prototyping board equipped with a Cortex M3 microcontroller. We employ dummy controllers void of any concrete logic and benchmark the latency of the single controller switch using either activation type and with or without stated deadlines.

According to our measurements, on average, *lazy* activation incurs in 7.7μs (6.8μs) latency with (without) the *deadline* option, whereas the *fast* activation requires 6.7μs (5.6μs). As expected, lazy activation is slower, whereas fast activation instantiates the new controller as soon as possible. Applying deadlines leads to additional processing time because of the implementation of the underlying semantics. Overall, the absolute numbers are very limited, and would hardly affect the timings of the application-level processing that typically ranges over orders of magnitude larger values.

3 References

[1] ArduPilot. www.ardupilot.com.

[2] T. R. Bretschneider and K. Shetti. Uav-based gas pipeline leak detection. In *Proceedings of the Asian Conference on Remote Sensing*, 2015.

[3] Cleanflight. www.cleanflight.com.

[4] OpenROV. www.openrov.com.

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

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