Abstract

We present our programming constructs that allow developers to gain a control on timing aspect in enforced adaptation decisions in real-time Cyber-Physical System (CPS) software. CPS operation often depends on the environmental conditions, and CPS software must adapt to the changes in these conditions. In CPS software a control-loop is run every 1-10 ms or even faster, forcing the adaptation to be limited with time boundaries. Our investigations have shown that developers are left without dedicated programming support, and the timing aspect of the adaptation has to be handled by the programmers manually. We argue that the time aspect is crucial in enforced adaptation decisions in real-time CPS software. Unlike the existing approaches, our language abstractions allow the developers to define the adaptation time boundaries and to avoid possible software misbehavior.

1 Overview

The natural way to program real-time CPSs is to create a dedicated controller that updates the commands for the system every 1-10 ms or even faster. From within the controller developers instruct the CPS how to behave according to the internal state of the CPS and data sampled from the sensors. The sensoric data the controller rely on may change drastically, as it depends on environmental conditions. The software must adapt to these changes, and the developers might need to switch to the completely different controller at runtime under time limits, as both the adaptation and the controller execution must complete within 1-10 ms. Our investigations have shown that the developers typically either invest a lot of efforts to gain a control on or do not pay much attention to timing aspect of the adaptation. With the latter approach the adaptation latency might be unacceptably large, while the former approach makes the software overcomplicated.

Example. Consider the Gas Leak Localization [2] task, where a set of aerial drones are locating a possible gas leak in an indoor environment. Each drone moves to the predefined location in the area. Upon arriving, drones use the Hovering controller where a drone is commanded to hover and to sample the gas concentration on its own position using the pre-installed gas sensor. Whenever the drone detects a high gas concentration, the system switches to the Leak Localization controller that broadcasts alert beacons via low-range radio. All the neighbors that received this beacon also switch to the Leak Localization controller and come closer to the drone with the highest gas concentration to get more fine-grained measurements of the area.

One of the possible implementations of such functionality is depicted in Figure 1. This is an approach similar to the one used in a real code-base of ArduPilot [1]. The function `set_flight_mode()` is called whenever the adaptation is required and, on lines 5 and 7, initializes the necessary controller. After successful initialization, the system executes a clean-up routine of the previous controller, as shown on line 9. The problem arises when the initialization and clean-up routines must be completed within the required time boundaries: as the developers left the timing aspect of the adaptation unattended, these routines may require much more time. The developers, however, did not consider this problem at all.

Moreover, we revealed a potential source of conflicts during the adaptation. For example, the developers typically ini-
In our work we address these issues and provide the following contributions:

- Our programming concepts allow the developers to gain a control on timing aspect in enforced adaptations that is usually missing in the adaptive real-time CPS software.
- We ease the burden of implementing the adaptation mechanism allowing the developer to choose between the programming efforts and adaptation latency.

Using our programming concepts, the developers can achieve the following benefits:

- Specifying a *deadline* the programmers gain a control on the timing aspect of the adaptation: whenever the deadline is violated, the error event is fired up allowing the developer to prevent the software from freezing.
- Our *activation types* bring two ways – lazy and fast – of the activation routine and allow the developers to choose the most appropriate one depending on the application. The lazy activation type avoids the conflicts between the controllers, but increases the activation latency. Differently, the fast activation type requires lesser time, but does not avoid possible conflicts.

We describe next how we implemented these concepts in a programming support, and what trade-offs the developers might expect using our activation types with and without the deadline.

## 2 Programming Concepts

We implemented our concepts in a number language abstractions and keywords for C++ language.

**Lazy activation.** This activation type can be invoked with the command `activate Hovering lazy`. With the latter the initialization of the Hovering controller is only executed after the clean-up routine of the previous controller, as displayed on the sequence diagram 2a. During this procedure, the system is in *Uncertainty* state, because neither of the controllers can be considered active. The interference is avoided, but the latency between the activation command and the notification is increased. As both the adaptation routine and the new controller execution must be complete within the time limits, the increased latency may negatively impact on the overall CPS operation. This is the price for the adaptation ease.

**Fast activation.** We introduce this activation type to deliver different activation latency vs. programming support trade-off. With the command `activate Hovering fast`, as displayed in sequence diagram 2b, the system initializes the Hovering controller before the clean-up routine of the previous one, reducing the latency between the activation command and the notification. As we argued before, this order may lead to the possible functionality conflicts. The latter, however, can be avoided with the additional programming efforts done by the programmers. For example, one may implement wrappers that prevent the execution of the asynchronous and periodic tasks during the adaptation. To be sure that this approach is feasible, we experimented with the target platform, as described in the next section.

### Deadline

A modifier *within*, being used after the activation command, determines the time boundaries for the adaptation. For example, in our scenario, the drone can not be out of the control for more that 10ms. Consider, the control processing requires 5ms, thus, the activation routine either lazy or fast should finish within 5ms. To this end, the programmer can use the command `activate Hovering within 5ms`. Should the activation process not finish within the 5 ms, the activation will stop, the previous controller will be initialized again, and the programmer will be prompted to handle the activation error.

## 3 Preliminary Results

To evaluate our approach we implemented a set of micro-benchmarks for Nucleo STM32L152 prototyping board – the platform similar to the typical platforms for aerial drones. Each benchmark calls a single adaptation command with certain parameters – such as activation types and deadlines – multiple times. This command activates mock-up controllers with empty bodies. Thus, the time required for execution of such activation command reflects the time needed only for activation routine. In our experiments we compare the adaptation time of our approach with its state-of-the-art counterpart, which is displayed in Figure 1.

In our experiments we provide the evidence, that the fast activation type is feasible from the programming efforts perspective. In doing so, we implemented wrappers – the software components that encapsulate any asynchronous operations – for the target platform. Any wrapper instance can be dedicated to a single controller, and, whenever the asynchronous task of the mode is about to be executed, the wrapper intercepts the command and executes it only if the controller is active. As we expected, the implementation of such wrappers is possible and feasible for our target platforms, such as Cortex-M3 based platforms.
Figure 3: The activation types deliver adaptation latency vs. programming efforts trade-offs.

Figure 3 depicts the adaptation latency the developer might expect using different activation types. The lazy activation type requires more MCU time to activate a control-loop, as expected. It is justified in the applications where developers heavily rely on the asynchronous operations and do not want to trade the performance of the standard classes with the safety delivered by the wrappers. On the other hand, one can get the faster adaptation by using the fast activation type, which is suitable for the applications where developers need the faster adaptivity. However, as the cost of the latter the developers must use the wrappers whenever any asynchronous operations are needed. In any case, the deadline option leads to the additional MCU performance overhead.

4 References