

# Non-disruptive Large-scale Component Updates for Real-Time Controllers

Michael Wahler   Stefan Richter  
Sumit Kumar

ABB Corporate Research  
Industrial Software Systems  
5405 Baden-Dättwil, Switzerland  
michael.wahler@ch.abb.com

Manuel Oriol

University of York  
High Integrity Systems Engineering  
Heslington, York, YO10 5DD, United Kingdom  
manuel@cs.york.ac.uk

## Abstract

Real-time controllers handle safety-critical environments such as power grids in a continuous way. Controllers are only updated during the same maintenance periods as the plant they control. As a consequence, old software versions are usually used longer than necessary, which can have a negative impact on performance, reliability, or security.

This paper presents a solution for updating component-based controllers at runtime in a non-disruptive way – there is no “bump” in the control cycle. This solution allows for simultaneous updates of real-time components with arbitrarily large states. The solution is validated by demonstrating a large-scale dynamic software update on an embedded controller with a 1 kHz control cycle on RT Linux. As a corollary, we show that an arbitrary number of components can be simultaneously updated in a non-disruptive way.

**Keywords** software update, real-time, non-disruptive, state transfer

## 1. Introduction

In the power and automation domain, plants are controlled by programs taking decisions in real-time and running on specific operating systems and hardware suitable for real-time execution. The lifetime of control systems can span over 30 years or more and their availability for safety-critical systems can be required to be up to 99.9999%. Over such large amount of time, the requirements for these systems can change, e. g., when alternative power generation technologies are integrated into an existing grid. Furthermore,

algorithms can evolve, bugs be found, and security fixes be issued. Applying such changes to the control system usually means stopping the plant while the update is applied. This often results in updates being delayed until the next maintenance period is scheduled, which may happen as rarely as once a year. This implies that a real control system would clearly benefit from having dynamic updates which do not disrupt its operation.

Existing solutions propose to modify the operating system (OS) to enable dynamic updates ([Gracioli and Fröhlich 2008]). This is not practical in our case because to achieve certifications as well as for business reasons, companies often use commercial off-the-shelf (COTS) operating systems (OSes) and cannot alter low-level mechanisms for the development of commercial highly-available systems.

We have previously presented a component-based solution on top of the commercial OS Integrity that copes with such issues [Wahler et al. 2009]. Our system relied on the assumption that all state transfers of any given component could be performed within one cycle of the schedule. While this is acceptable for toy applications, it does not cope with real-world programs which contain potentially large states. We have further refined our solution by introducing transactional ACID properties, significantly reducing context switches, and not relying on components to actively hand back their channels when updated.

In this article we present an improved solution, which allows to gradually build the state of the new component once an update has been announced and loaded. When the whole state is transferred to the new version of the component the update takes place as an atomic operation. The implementation of this solution runs on any POSIX-compatible OS, e. g., RT-Linux.<sup>1</sup>

Section 2 presents the component framework and explains its internals. Section 3 details the update mechanism. Section 4 evaluates empirically the approach. Section 5 presents related work and we finally conclude in Section 6.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

HotSWUp'11, April 16, Hannover, Germany.  
Copyright © 2011 ACM ISBN .....\$10.00

<sup>1</sup><http://www.rtlinuxfree.com/>

## 2. Component Framework

Our framework relies on four structural concepts: component, block, port, and channel. In this section, we describe these structural concepts and the runtime concerns in detail.

### 2.1 Components and Blocks

A component is a piece of software that has a high cohesion: a component’s functionalities are strongly related to each other. It contains code and an optional *state*, which is used for memorizing data. The code within a component is structured into *blocks*. Blocks represent the behavior of a component and form the atomic units of execution, i. e., the execution of a block cannot be interrupted or preempted by other blocks. A component must contain at least one block and can contain arbitrarily many. Only a component’s blocks can access its state.

Figure 1 shows an example application for overvoltage protection. This application has three components *sensor*, *overvoltage*, and *actuator* with one block of execution each. In each cycle, component *sensor* is executed first and acquires measurement data, e. g., via IEC 61850-9-2 sampled values. This data is passed to the component *overvoltage* where it is stored – along with a history of past values – in its state. If *overvoltage* detects an abnormal situation, it sends an operate signal to *actuator*. When the component *actuator* receives such a signal, it sends a trip signal to a circuit breaker via I/O.

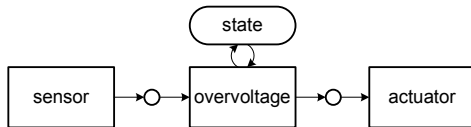


Figure 1. Example application

Encapsulating a component’s behavior in blocks has several advantages. First, temporal dependencies between blocks are made explicit when a system is engineered. No additional synchronization is required, which could introduce unnecessary idle time or even deadlocks otherwise. Second, the number of context switches is reduced because there is no interleaved block execution (blocks can only be executed in parallel on multi-core CPUs). This reduces CPU load, which is particularly important in embedded systems. Third, a block is required to run to completion once it is started, e. g., its code must not contain infinite loops. This is an important prerequisite for deterministic system behavior. Fourth, components can have standard blocks. As an example, every stateful component has two standard blocks *teach* and *learn*, which will be introduced in Section 3.

### 2.2 Ports and Channels

A block’s interface is composed of an arbitrary number of typed input and output ports. Through their input ports, blocks receive parameters for each execution; the result of

each block execution is published in the block’s output ports. Input and output values are only processed in the cycle in which they are transmitted.

In order to receive input values, input ports need to be connected to output ports via channels. Channels are unidirectional and connected to exactly one input port and one output port. In addition, an output port of a block B1 may only be connected to an input port of a block B2 if B1’s execution finishes before the beginning of B2’s execution. This implies that all systems constructed from channels and blocks can be described by an acyclic, directed graph with blocks as nodes and channels as edges. In our example application (Figure 1), there are two channels (visualized by circles) connecting the ports (visualized by arrows).

### 2.3 Runtime

Modern real-time operating systems offer schedulers for the concurrent execution of multiple threads [Baker and Shaw 1989, Locke 1992]. Often, schedulers allow to add threads dynamically to a running system. Preempting a thread to resume another requires a context switch in the CPU, which costs computation time and thus reduces performance. Such dynamic scheduling is not needed for cyclic control systems because the system structure rarely changes. Scheduling follows a fixed order in which components are executed. Therefore, our framework uses a static but reconfigurable schedule for the blocks that are executed cyclically.

The scheduler of the framework, which runs in an OS thread at the highest priority, executes the schedule of blocks in a cyclic fashion at a given base frequency. To this end, timer interrupts are generated at regular intervals, each of which triggers one execution of the scheduler. In Figure 2, timer interrupts are displayed as vertical arrows. The scheduler then calls some or all blocks according to its schedule, which is represented by the gray bars in Figure 2. After the last block has been executed, the scheduler waits until the next timer interrupt. During this period of time, which we call *slack time* (represented as white bar), other software (e. g., an FTP server) may run asynchronously in a different OS thread. This software will be preempted by the operating system as soon as the timer interrupt triggers the scheduler for the execution of the next cycle.

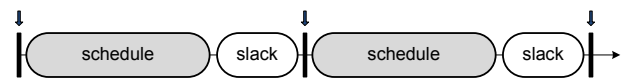


Figure 2. Timeline of schedule execution

The schedule for the application in Figure 1 is shown in Figure 3(a). As can be seen, the three blocks are executed sequentially every 1 ms (thus, the cycle frequency is 1 kHz). The runtime overhead of using our component framework is around 0.5  $\mu$ s per block. Thus, several hundred blocks can be scheduled in a typical control cycle of 1 kHz.

A system *configuration* comprises a schedule and a set of pointers to the required blocks and channels, respectively.

At any point in time, there is exactly one active system configuration.

### 3. Updating Mechanism

Updates in our system consist of three steps. First, the new component is instantiated and initialized, new channels are created if necessary, and a new schedule is created in which the blocks of the old component are removed and to which the blocks of the new component are added. Second, the state of the old component is transferred to the new component.

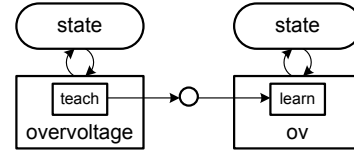
This transfer takes into account that the old and the new component may reside in different address spaces. Therefore, it is not sufficient to transfer a pointer to the state of the old component, but instead, all the data in the state must be transferred. Our update mechanism provides for transferring very large states, which cannot be transferred in the slack time of a single cycle. Since the state of the old component may change during this transfer, we implement a *state synchronization* algorithm that tracks such changes. These steps are performed during the slack time of the control cycles and can last for an arbitrary number of cycles, i. e., they do not need to meet real-time deadlines. Once the states of the new component and the old component are synchronized, the system instantaneously switches from the old configuration to the new configuration at the end of a control cycle. This switch consists of a single pointer change, which allows for atomic updates of multiple components.

In this section, we illustrate how a component can be dynamically updated in our system by replacing the component `overvoltage` with an updated version called `ov`.

#### 3.1 Component Update

We assume that the system executes its original configuration, i. e., the application from Figure 1 with the schedule shown in Figure 3(a). When the update is initiated, the framework loads and instantiates the code of the new component `ov`. Afterwards, the framework creates a new channel, which connects the `teach` block from `overvoltage` and the `learn` from `ov` as shown in Figure 4. Once this channel has been created, these two blocks are added to the schedule as shown in Figure 3(b) and start to execute the state synchronization algorithm, which is explained in the subsequent section. When the state synchronization is finished, the `teach`

block notifies the framework, which switches to the updated schedule Figure 3(c) before the start of the next computation cycle and thus ensures an atomic update.



**Figure 4.** Teach and learn blocks during state synchronization

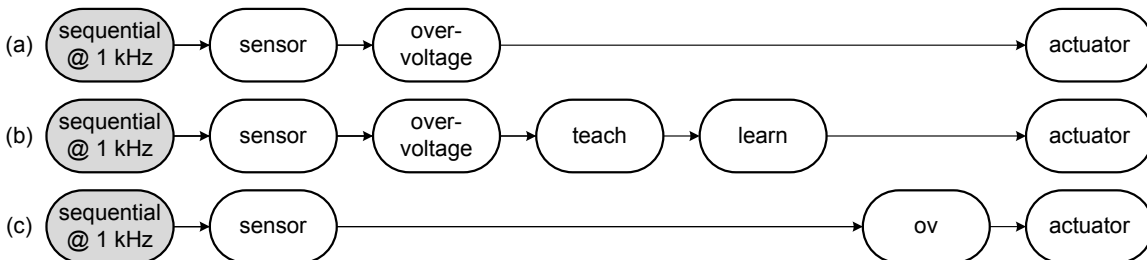
This update mechanism supports a rollback of the system because old configurations can be kept in memory. In case a new configuration does not behave as expected, an old configuration can be restored with another atomic switch.

Our component framework also enables lightweight updates by replacing blocks in components and leaving the state unchanged. This requires that the updated software must use the exact same data structures in the state. In the following subsection, we present a sophisticated state synchronization algorithm that allows the state of the original and the updated component to be structured differently.

#### 3.2 State Synchronization Algorithm

For state synchronization, we implemented the `TEACHING` algorithm from [Steiger 2008]. This algorithm distinguishes between teacher, i. e., the old component, and learner, i. e., the new component, which replaces the old component. During multiple control cycles, the teacher transmits its state to the learner. As soon as the complete state has been transferred (i. e., the states of teacher and learner are synchronized) the teacher component is replaced by the learner.

The algorithm takes into account that the teacher’s state can change during the synchronization. To this end, the `TEACHING` algorithm uses a *dirty bit* vector for memorizing which parts of the state have already been synchronized. This algorithm assumes that the state can be divided into  $n$  parts and thus managed by a bit vector of length  $n$ . It further assumes that the rate at which the state is transferred is higher than the rate at which the state changes. As an example, the algorithm terminates if per cycle, 3 kB of the state change, but 4 kB can be transferred to the learner.



**Figure 3.** Schedule before (a), during (b), and after (c) the update

When teaching starts, all dirty bits are set because no part of the state has been synchronized. The first message sent by the teach block contains the version number of the component. In each cycle, a non-empty portion of the state is transferred from the teacher to the learner. The respective bits in the dirty bit vector are cleared.

If the state of the teaching component changes, the respective dirty bits are reset, which ensures that parts of the state that have already been transferred are transferred again after they have changed. Eventually, all dirty bits will be cleared and the teacher signals to the framework that teaching is finished. Subsequently, the teach and learn are removed from the schedule as shown in Figure 3.

The state synchronization algorithm allows the old and the new components to use different data structures for the state. As an example, the state of `overvoltage` could be an array of type `double` and size 1024 kB. The state of `ov` could be an array of type `int` and size 2048 kB. The teach block of `overvoltage` will send messages containing an (index, value) tuple each. When `learn` receives a message  $(i, x)$ , it will set its state at index  $2 \cdot i$  and index  $(2 \cdot i) + 1$  to  $x$ .

If the state is too volatile, this strategy might delay updates indefinitely. To avoid that, users can define their own policy for transferring the state. When this is relevant, we always transfer the more long-lived part of the state first, leaving the more volatile part for the end. We are currently investigating a way of generating automatically this ordering in an efficient way.

Despite such sophisticated solutions, we would like to state that there is no generic solution for state synchronization. There will always be cases in which the state changes so frequently that it cannot be consistently transferred across multiple cycles. One example is a component for parsing sampled measurement values according to the standard IEC 61850-9-2. The state of this component contains a buffer on which operations such as resampling are performed. The buffer changes in every cycle and since it can be several hundreds of kilobytes in size, there is no means for synchronizing it with another component over multiple cycles. For such cases, different means of ensuring that the state of the old and the new component are consistent must be used. As an example, old and new component may operate in a “work-by” mode [Steiger 2008]. In this mode, both components receive the same input in each cycle, but the output ports of the new component remain unconnected. Once the components have operated long enough in parallel such that their states are equivalent, the system atomically disconnects the old component and connects the output ports of the new component to the system. This requires however that the load of the system is sufficiently low to schedule the blocks of both components simultaneously.

## 4. Validation

In this section, we validate our real-time updating mechanism by updating the component `overvoltage` with component `ov` at runtime and measuring the impact on the system’s behavior.

### 4.1 Setup and Criteria

When updating, we are interested in the resulting jitter, i. e., the deviation of the actual cycle frequency from the ideal cycle frequency. In real-time systems, the jitter should be as low as possible. The most typical causes for jitter are interrupts – e. g., by hardware devices –, context switches by the operating system – e. g., when multiple threads are executed in parallel –, or imprecise system timers.

The platform for the validation is a Freescale Lite5200B development board containing a single-core PowerPC processor running at 396 MHz and 256 MB of memory. As operating system we use RT-Linux. The Linux kernel used is 2.6.16.11-rt18 with a timer frequency of 1 kHz. On the target system, the example application (cf. Figure 1) and the measurement instrumentation have an execution time of about 0.75 ms, i. e., there are 0.25 ms of slack time available in which the states can be synchronized and the configurations be changed. This amount of slack time allows us to synchronize 9 items of the state in one cycle on the validation platform.

The validation platform is a multi-threaded system and uses Ethernet and serial I/O. Therefore, there will be a certain amount of jitter in the system. We first measure this jitter during the execution of the example application (Figure 1) without updates:

(M1) The *baseline*: minimum, maximum, and average jitter between cycles during normal system activity

Analogously, we measure the system jitter while updates are performed.

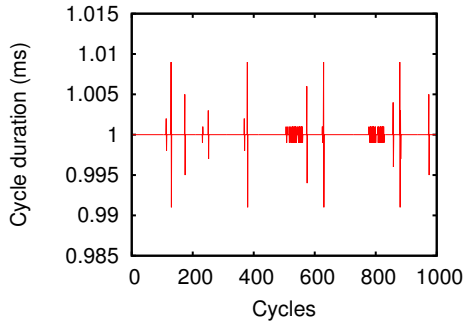
(M2) Minimum, maximum, and average jitter between cycles during the software update.

### 4.2 Results

As the baseline (M1), we measured the cycle time during the normal system activity. Figure 5 shows a graph with the cycle duration in seconds during the first 1000 cycles of the system. As can be seen, there is zero jitter for most cycles (i. e., the cycle duration is exactly 1.00 ms). Periodically, the jitter reaches around 1% of the cycle duration and there are phases during which the jitter is around 0.5%. Since no updates are performed, this jitter is caused by system activities such as interrupts by the network interface.

In our experiments, we measured a maximum jitter of 30.00  $\mu$ s, which is around 3% of the ideal cycle duration. Table 1 shows all results of the measurement.

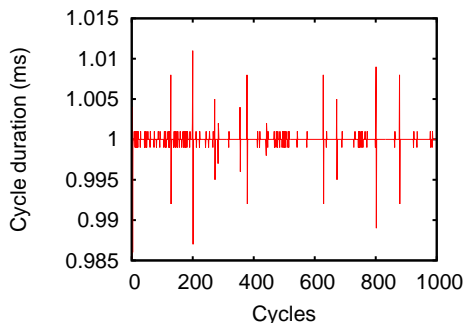
We set up the validation for (M2) such that the original system runs according to the schedule shown in Fig-



**Figure 5.** Cycle times during normal system activity (M1)

ure 3(a) for 160 cycles. Subsequently, the update is started by scheduling the teach and learn blocks for 600 cycles. Subsequently, teach, learn, and overvoltage are removed from the schedule and the latter is replaced with ov. After this, the updated system runs for another 200 cycles before the test terminates and writes the measurement values into a file.

Figure 6 shows the graph of the corresponding cycle durations. It can be seen that the reconfiguration process running in the background frequently causes a very small jitter (less than 0.2%). Periodically, the jitter reaches around 1% as in the measurements (M1).



**Figure 6.** Cycle times during system update (M2)

Table 1 shows all measurement results for jitter during normal system activity (M1) and dynamic updates (M2). As can be seen from this table, the average jitter is the same in (M1) and (M2). The maximum jitter is in the same order of magnitude. Thus, no bumps are introduced by the dynamic software update.

%	min	avg	max
(M1)	0.00	0.03	3.00
(M2)	0.00	0.03	4.70

**Table 1.** Jitter in percent during normal system activity (M1) and update (M2)

## 5. Related Work

[Vandewoude and Berbers 2002] contains a detailed study of several approaches to dynamic updates and concludes that none of these approaches are directly applicable to updating real-time systems. As well as in [Gracioli and Fröhlich 2008, Soules et al. 2003, Wahler et al. 2009], the study points out that component-based architectures are, however, particularly well-suited for updating real-time systems.

The inherent difficulty with dynamically updating real-time systems is the need for predictability and some systems go as far as requiring all versions of the components to be present from the start on the device to reconfigure [Stewart et al. 1997]. Another example [Calvo et al. 2010], presents an approach for dynamically reconfiguring a system based on a real-time variant of CORBA. This approach uses a networked video system as a case study and relies on a loose coupling between the components and timing aspects of the underlying network. Dynamic updates of the components involved are not addressed.

In [Zoitl et al. 2010], a modular software architecture is presented that allows for dynamic reconfiguration of real-time control systems running at a high cycle frequency (200 Hz). In contrast to our approach where updates can be prepared in an arbitrary amount of time, the reconfiguration must meet real-time deadlines. It is not mentioned in [Zoitl et al. 2010] if internal states of components can be preserved when components are updated.

Another difficulty is that updates also typically degrade the system’s performance, thus potentially causing deadlines to be missed [Segal and Frieder 1993]. As well as for the previous difficulty, our system copes with it explicitly.

## 6. Conclusions and Future Work

We have presented a method for dynamically updating components in a real-time system. The update preparation phase can consume an arbitrary, but finite amount of time, enabling the migration of the state from the old to the new version of the component even if the data structures differ significantly.

Because the time-consuming part of the update is distributed over several cycles, what we presented does not only cope better with updating components with states of arbitrary size, but also an arbitrary number of components can be updated simultaneously. Validating further this fact is still future work.

Future work also includes an automated analysis of the amount of slack time available and an adaptation of the rate at which states are synchronized – at the moment, this information is hard-coded.

The biggest challenge for updating the software of real-time systems is that they often need to be certified according to standards such as IEC 61508.<sup>2</sup> If the software of such a system is updated, the system as a whole must be re-certified

<sup>2</sup><http://www.61508.org>

before it can be used in the field. We are involved in the EU project PINCETTE,<sup>3</sup> which aims at the automatic detection, localization, and repairing of bugs for intra-component changes and component replacement.

## References

- T. Baker and A. Shaw. The Cyclic Executive Model and Ada. *Journal of Real-Time Systems*, (1), 1989.
- Isidro Calvo, Luis Almeida, Federico Perez, Adrian Noguero, and Marga Marcos. Supporting a Reconfigurable Real-Time Service-Oriented Middleware with FTT-CORBA. In *IEEE International Conference on Emerging Technologies and Factory Automation*, 2010.
- G. Gracioli and A. A. Fröhlich. An Operating System Infrastructure for Remote Code Update in Deeply Embedded Systems. In *First Workshop on Hot Topics in Software Upgrades (HotSWUp'08)*, 2008.
- C. Douglass Locke. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. *Journal of Real-Time Systems*, (4):37–53, 1992.
- M.E. Segal and O. Frieder. On-the-fly Program Modification: Systems for Dynamic Updating. *IEEE Software*, 10(2):53–65, 1993.
- C.A.N. Soules, J. Appavoo, K. Hui, R.W. Wisniewski, D. Da Silva, G.R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System Support for On-line Reconfiguration. In *Proceedings of the 2003 USENIX Technical Conference*, pages 141–154, 2003.
- Michael Steiger. Fault-Tolerant Turbine Controller. Master's thesis, ETH Zurich, 2008.
- D.B. Stewart, R.A. Volpe, and P.K. Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, 1997.
- Y. Vandewoude and Y. Berbers. An Overview and Assessment of Dynamic Update Methods for Component-oriented Embedded Systems. In *Proceedings of The International Conference on Software Engineering Research and Practice*, pages 521–527, 2002.
- Michael Wahler, Stefan Richter, and Manuel Oriol. Dynamic Software Updates for Real-time Systems. In *Second International Workshop on Hot Topics in Software Upgrades*, 2009.
- Alois Zoitl, Wilfried Lopuschitz, Munir Merdan, and Mathieu Vallee. A Real-Time Reconfiguration Infrastructure for Distributed Embedded Control Systems. In *IEEE International Conference on Emerging Technologies and Factory Automation*, 2010.

---

<sup>3</sup><http://www.pincette-project.eu>