Dynamic Software Updates for Real-Time Systems

Michael Wahler Stefan Richter
ABB Corporate Research
Industrial Software Systems
5405 Baden-Dättwil, Switzerland
michael.wahler@ch.abb.com
stefan.richter@ch.abb.com

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

Abstract

Seamlessly updating software in running systems has recently gained momentum. Dynamically updating the software of real-time embedded systems, however, still poses numerous challenges: such systems must meet hard deadlines, cope with limited resources, and adhere to high safety standards.

This paper presents a solution for updating component-based cyclic embedded systems without violating real-time constraints. In particular, it investigates how to identify points in time at which updates can be performed and how to transfer the state of a component to a new version of the same component. We also present experimental results to validate the proposed solution.

Keywords dynamic software update, real-time, embedded

1. Introduction

In the power and automation industry, embedded systems control and protect *primary equipment*. The physical tasks performed by these pieces of primary equipment are often expensive to interrupt. Stopping big automation processes is usually expensive due to loss of production and considerable start-up cost. For instance, breaking operation of a power substation can imply power outage for customers.

As a consequence, the software of such embedded systems is hardly updated during its lifetime, which can span over 30 years and more. Updates only take place at rare points in time, usually at scheduled shutdowns of the primary equipment. There are three main causes for updating such systems dynamically. First, systems have to be adapted to newer standards of communication protocols such as IEC 61850 or for increased security requirements such as IEC 62351. Second, as engineering know-how advances, operators might want to incorporate new protection or optimization algorithms into their running installations. Third, as newer software becomes increasingly complex, critical bugs are more likely to occur.

There are several solutions to updating software dynamically; few of these have actually been adapted to real-time systems, which need to complete tasks at hard deadlines. As an example, in protection of power networks, a short circuit on a medium-voltage line must be detected within 20 ms. It is critical to always meet deadlines, even when updating the software. It is thus essential to identify points in time at which an update can be performed, and the update must finish in a deterministic amount of time. This should also account for transferring the state from the replaced module to the replacing module if possible and required.

Some of the existing solutions propose to modify the operating system (OS) to enable dynamic updates ([Gracioli and Fröhlich 2008]). However, to achieve certifications and 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.

This paper presents how to perform dynamic software updates on a real-time system that runs a COTS operating system. Our solution relies on using a component-based infrastructure in which components communicate through a framework layer and can be replaced individually or in group at runtime. To perform an update in real-time, we rely on spare time available after the code actually executed and before the cycle finished, possibly distributing an update on several cycles if necessary.

We first identify features required for the OS and present a framework that enables dynamic software updates on a COTS OS in Section 2. Section 3 shows how to identify points in time to apply real-time software updates and elaborates on the problem of updating *stateless* and *stateful* components in real-time. Section 4 presents experimental results.

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'09, October 25, 2009, Orlando, Florida, USA.
Copyright © 2009 ACM ISBN 978-1-60558-723-3/09/10. . . $10.00

Section 5 compares our solution to related work. We summarize our findings in Section 6.

2. Platform

Updating software relies on a few specific features of the underlying software platform, including the OS. For liability and long-term support, our requirements specify that using a commercial OS is mandatory. This makes it practically impossible to modify internal structures of the OS as, for example, done by [Gracioli and Fröhlich 2008].

Modern real-time OSes nevertheless provide features that indirectly support dynamic software updates. Subsection 2.1 evaluates three modern real-time OSes. Subsection 2.2 presents our component-based framework, which supports the replacement of components in real-time.

2.1 Operating System

In [Wahler and Richter 2009], we evaluated three real-time OSes with respect to dynamic software updates: Wind River VxWorks in version 6.6, Green Hills Integrity 5.0.11, and QNX Neutrino 6.3.2. By default, VxWorks is the only OS in which software can be updated dynamically by replacing single C functions. While this update mechanism offers a fair amount of flexibility, it is not usable for high-performance real-time systems because such updates can lead to unbounded interruptions of the system, lasting for several seconds.

Although none of the evaluated OSes offer real-time software updates, all of them offer mechanisms sufficient to build a framework capable of dynamic updates: *remote debugging* allows system engineers to load code onto a running system, terminate processes, and start new processes; a *message passing interface* provides indirection for the communication (and thus, independence) between different parts of the software; *separation* mechanisms such as separate address spaces for threads provide additional independence. These properties allowed us to build a component-based framework infrastructure on top of the OS, for which we chose Integrity [Wahler and Richter 2009].

2.2 Component Framework

Our component framework relies on a strict separation of components as well as indirect means of communication between components. The two main concepts of the framework are *components* and *channels*. Each component executes as a process in a separate address space, which entails strict separation. Thus, failures in one component do not arbitrarily propagate throughout the system although it is always possible to propagate invalid values to other components [Tai et al. 2002].

Components are software modules that offer well-defined services via input and output ports. The computation of a component must terminate on any kind of request. Since address spaces are separate, components communicate indirectly via message passing. Channels are buffered connections between components. The *component manager*, which is an updateable component itself, creates and maintains the channels and can dynamically rewire them while the system is in operation.

Applications are function plans, i. e., data-flow networks in which components are the nodes and channels are the edges. Similar to [Magee et al. 1989], components in our framework can be connected dynamically, i. e., connections between components can be added or removed dynamically.

Power protection devices are a class of systems which are executed cyclically. To simplify the presentation in this paper, we assume that exactly one cycle with a fixed frequency is executed in a system. In Figure 1, we present an example of a function plan to which we refer in the remainder of this paper. Read from left to right, each cycle starts at a point s, which is cyclically triggered by the component manager at a fixed frequency. Next, an input signal is received by component `Sensor`, which carries out its computation and provides an output. Finally, the component manager triggers the execution of `History`, which reads the output of component `Sensor` and stores this data.

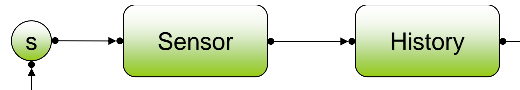

Figure 1. Example function plan

The component manager is responsible for the application to meet its deadline. In the above example, assume that the cycle frequency is 200 Hz, i. e., the component manager triggers the execution of a cycle every 5 ms. The component manager knows the worst-case execution times of all components, e. g., 200 µs for `Sensor` and 100 µs for `History`. If a fault causes the component `Sensor` to not terminate within 200 µs, the component manager can abandon the execution of the faulty component, send a default value to `History`, and start exception handling.

The framework is implemented in C++ and uses template classes to a large extent to establish a high degree of abstraction and thus, a high degree of portability to other OSes.

3. Real-Time Updates

This section presents how to update components in our framework. First, we identify a point in time at which to perform updates. Second, we provide an algorithm for loading new components and handing over control from an old component to a new component. We discuss state transfers separately in Subsection 3.3.

3.1 Scheduling Updates

There are several points in time at which a component could be updated. For instance, each component could check for an update before or after it is executed. We decided to let the system perform such checks at the end of each cycle for

all components because then, they occur at a time at which the CPU would be idle anyway (waiting for the start of the next cycle) and thus have minimal impact on the system's behavior. In our example in Figure 1, the component manager informs components that updates are available after the execution of `History`.

If multiple components need to be updated, it is possible that the time available for updates per cycle is too short for performing all updates. In this case, the updates of single components have to be distributed across multiple cycles if this does not violate the ACID transactionality principles. The OS typically ensures these principles when only one component is updated because the whole component must be uploaded before it can be started (atomicity), the uploaded component must be a valid binary image (consistency), the upload does not interfere with other processes (isolation), and once a component has been updated, it remains persistent (durability). In case the update cannot be fragmented over several cycles, we follow an "all or nothing" policy. If transactionality cannot be established (e. g., when a state is too large to be transferred within one cycle), additional techniques should be applied such as state convergence algorithms, but this goes beyond the scope of the present article.

3.2 Update Algorithm

To perform updates, there must be enough free memory in the system to load the new component. Furthermore, the CPU load caused by the main application must be sufficiently low to allow the framework to load a new component while executing the main application.

The following steps must be executed in order to update a component c with a component c'.

1. The new version c' of component c is sent to the target machine using the OS functionality. The transmission happens with low priority, in (quasi-)parallel to the running application.
2. This upload is detected by the component manager, which initiates the replacement of c.
3. The component manager tells the old component c to exit.
4. The component c hands the connections to its channels back to the component manager and terminates.
5. The component c' acquires the connections that were previously owned by c from the component manager and starts its operation.

While the steps 1 and 2 must not meet any deadline (in fact, they can span multiple cycles), the steps 3 to 5 must be executed in real-time (they must be finished before the end of the cycle). To ensure that these steps are executed deterministically, component developers should not alter them. In the framework's generic superclass `Component` they are implemented as private methods. The time required for the handover is linear with respect to the number of connections of the updated component.

An advantage of our approach is that after an update is complete, there is no overhead in the running system due to updates. As a consequence, our systems can be updated arbitrarily often without a gradual decrease in performance. This is in contrast to systems in which multiple levels of indirections are required, which can increase execution times in an unbounded way.

3.3 Real-Time State Transfer

The previous section presented how components can be replaced in real-time, but omitted to describe how to perform state transfers. There are two main challenges for state transfer in real-time. First, the state should be transferred within the same cycle in which the handover from the old to the new component takes place — for example, after step 5 of the update algorithm in the previous section.

Second, if data types have changed in the new version of the component, the parts of the state that require a new format must be converted (e. g., by type wrapping [Neamtiu et al. 2006]). As an example, the old version of a temperature measurement component may store the temperature in degree Celsius and the new version may store it in Kelvin. This requires that 273.15 be added to each value in the state.

Developers must make sure that state conversion functions terminate in time and behave deterministically. It is difficult to propose design rules for state conversion functions. Their complexity should be as low as possible, but on the other hand, if the size of the state is particularly small, an exponential state conversion function could still be executed in the time slice of the component. Thus, the worst-case execution time of the state conversion function must be assessed experimentally.

In our component framework, state transfers are implemented as follows:

1. For each component instance, the component manager creates an area of shared memory of a predefined size. This memory area is labeled as non-initialized.
2. When a component starts, it acquires a pointer to this memory area from the component manager.
3. Before a component terminates, it writes its state to this memory area, marks the area as initialized, and adds its version information to the meta-data of the memory area.
4. When a component starts and acquires an initialized memory area, the component reads the version information and assesses whether its data types have changed from this previous version. If not, it copies the data from the memory area to its own state. If the data types have changed, it applies the appropriate state conversion function, which must be defined in the new component for its previous versions.

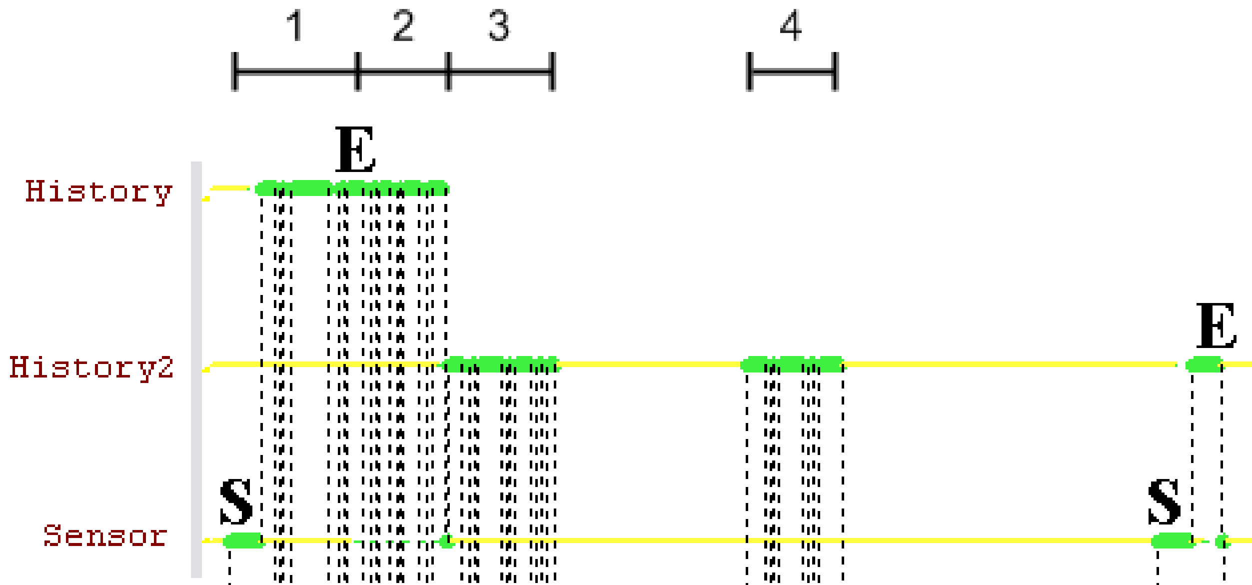

Figure 2. Updating component `History`.

We currently assume that the state can be transferred within the same cycle in which the component is updated. If this is not possible, converging state transfer algorithms are required, which we consider future work (see Section 6).

4. Experimental Results

We validated our approach by performing dynamic software updates for the example application shown in Figure 1. In this section, we present the results of our validation.

4.1 Setting

In the test application, at each cycle, the component `Sensor` calls the system clock to get the current time. The component `Sensor` sends this value to component `History`, which has a cyclic buffer to store a component `History` of the last 1000 values received. Since it stores each value as a 32-bit floating point number, the state of component `History` is 4000 byte large.

The annotated screenshot in Figure 2 shows various events generated during the execution of the application in the Integrity Event Analyzer, a tool for displaying processes and the events they generate. The letter *S* indicates the start of a computation cycle and the letter *E* indicates its end. Each cycle takes 273 µs to complete, which corresponds to the interval (1) between the events *S* and *E*. Our application runs at 200 Hz, resulting in a cycle duration of 5 ms, which corresponds to the interval between the two subsequent *S* events. As an example, in a 50 Hz power network, our system can complete four computation cycles during one period.

The evaluation machine used in our experiments has a MPC5200 processor running at 396 MHz with 256 MB of memory. We carried out two experiments. First, we updated component `History` to component `History2` without transferring the state. Second, we performed the same update including a state transfer from the old to the new component.

4.2 Measurements

Figure 2 shows the events generated during the update from component `History` to component `History2`. It can be seen that during interval (2), component `History` requires time to pass its channel connections and its state to the system. During intervals (3) and (4), component `History2` receives these resources from the component manager, which itself becomes active between interval (3) and (4) (not shown in the figure). It turned out that during the update, the system meets its 5 ms deadline. Thus, we can update the component without any impact on the real-time behavior of our system.

We have seen that dynamic updates of components are possible when we ignore their state. Next, we executed the update again and enabled state transfer from the old to the new component. We have validated that our system meets its 5 ms deadline even when a 4000 byte large state is transferred. We are currently exploring the size limits of state transfers in our framework.

The maximum cycle frequency that we could establish in the current implementation is about 300 Hz. System performance is limited by three factors. First, updating a component requires numerous context switches in our current implementation, as indicated by the vertical dashed lines in Figure 2. Second, most time during an update is spent on disconnecting and reconnecting channels; transferring the state requires only little additional time. Third, for safety reasons, we execute components in separate address spaces, which requires the use of message passing for communication. In comparison to direct function calls, message passing is about

80 times slower on our test system. As a consequence, future versions of our component framework will focus on optimizing resource usage. We are confident that cycle frequencies can be increased by at least one order of magnitude while still supporting dynamic software updates.

5. Related Work

There are two major types of approaches for updating programs dynamically: routine-based updates and component-based updates.

Routine-based updates are fine-grained and allow for updating individual routines. For example, the updating approach [Neamtiu et al. 2006, Stoyle et al. 2007] updates C-programs with a function-level granularity. The data, such as data from named types, is converted on demand when functions need to access it. This type of approach needs support from the runtime and necessitates code analysis in order to be performed in a type-safe manner.

Component-based updates are more coarse-grained and allow components to be updated at runtime. The communication layer is however more rigid as it guarantees isolation between components. The K-42 [IBM 2009, Soules et al. 2003] and the EPOS [Gracioli and Fröhlich 2008] OSes are such systems: they allow for updates of kernel objects/components at runtime.

The behavior of component-based systems can generally be adapted at runtime by rewiring the connections between the components, a technique referred to as *online reconfiguration*. Wang *et al.* [Wang and Shin 2002] for example present a system that can be reconfigured at runtime, but needs to be in a quiescent state (i. e., a state where the system does nothing) to perform the modifications. Obtaining component-based systems that perform updates at runtime requires transferring the state from old versions of the components to new ones when loaded, as for example LuckyJ [Oriol and Serugendo 2004]. Our approach falls into the last category and allows fully flexible updates of the components in a system in real-time.

While systems that support dynamic updates are flourishing, systems that also cope with real-time deadlines are few. [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. The study also concludes that component-based architectures are particularly well-suited for updating real-time systems, which is also confirmed by other authors later on [Gracioli and Fröhlich 2008, Soules et al. 2003] as well as in the present article.

The inherent difficulty with 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]. Updates also typically degrade the system's performance, thus potentially causing deadlines to be missed [Segal and Frieder 1993]. Our system copes with these two issues explicitly. The whole real-time control software is replaced in [Sha 2003]. The focus of this approach lies on fault prevention in the updated software. [Toy 1998] uses mechanisms provided by the UNIX RTR OS for updating the software of a real-time electronic switch system. This paper provides an in-depth description of the infrastructure necessary for updating highly available real-time systems.

The systems most similar to ours were designed more than 10 years ago [Alonso and de la Puente 1993, Alonso et al. 1995, Stewart et al. 1997].[1] [Alonso and de la Puente 1993] presents an approach for dynamic software updates in real-time systems in which whole processes, which communicate via a custom-made message-passing mechanism, can be replaced at runtime. During each update, the process to be replaced can send its state to the replacing process. The authors do not elaborate on the safety of the updates related to safe time points at which updates can take place. The behavior of the system is also unclear when there is not enough time to transfer the state of a component to the replacing component. In [Alonso et al. 1995], the authors only partially validated their approach because no RTOS that supported dynamic loading of code at runtime was available to them. Thus, all components that replace others must already be present on the target system as in [Stewart et al. 1997]. The cycle frequency in their proof of concept is 5 Hz (i. e., the process has a period of 200 ms). In contrast to this work, our validation is more extensive because modern operating systems allow operators to dynamically load code via a debug connection.

6. Conclusions and Future Work

This paper presents a component framework for real-time systems, which allows for dynamic software updates without missing real-time deadlines. The framework solely uses features common to commercial off-the-shelf OSes to update the code and relies on memory protection of processes to ensure separation. In this infrastructure, components communicate through a message passing mechanism based on channels. We show that transferring the state from the old version to the new version of the component is possible in real-time and we have validated our solution with a prototypical implementation.

We have identified three main avenues for future work. First, we must ensure that the ACID principles for transactionality (Atomicity, Consistency, Isolation, Durability) are obeyed when more than one component is replaced simultaneously as discussed in Subsection 3.1.

Second, we currently assume that components are good-natured, i. e., they hand back their resources and terminate when they are asked to. Although this behavior is implemented in the generic superclass of all components, it is still

[1] Interestingly enough, the main study on dynamic updates [Vandewoude and Berbers 2002] did not consider such systems.

possible to develop malicious components that do not terminate. In the future, the component framework should be able to revoke resources from components without their active participation, assuming that components can be malicious.

Third, the current implementation of our component framework requires many (assumedly unnecessary) context switches between threads, especially in the course of an update. This results in a performance penalty preventing us from reaching a higher frequency than 300 Hz. Future optimizations will focus on eliminating most context switches.

References

A. Alonso and J. A. de la Puente. Dynamic Replacement of Software in Hard Real-Time Systems. In *Fifth Euromicro Workshop on Real-Time Systems*, pages 76–81, 1993.

A. Alonso, I. Casillas, and J. A. de la Puente. Dynamic Replacement of Software in Hard Real-Time Systems: Practical Assessment. In *Seventh Euromicro Workshop on Real-Time Systems*, pages 26–33, 1995.

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.

IBM. The K42 Project. `http://www.research.ibm.com/K42`, July 2009.

J. Magee, J. Kramer, and M. Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, 1989.

I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical Dynamic Software Updating for C. *SIGPLAN Not.*, 41(6):72–83, 2006.

M. Oriol and G. Di Marzo Serugendo. A Disconnected Service Architecture for Unanticipated Run-time Evolution of Code. *IEE Proceedings-Software, Special Issue on Unanticipated Software Evolution*, 151(2):95–107, April 2004.

M.E. Segal and O. Frieder. On-the-fly Program Modification: Systems for Dynamic Updating. *IEEE Software*, 10(2):53–65, 1993.

L. Sha. Upgrading Real-Time Control Software in the Field. *Proceedings of the IEEE*, 91(7):1131–1140, 2003.

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 Online Reconfiguration. In *Proceedings of the 2003 USENIX Technical Conference*, pages 141–154, 2003.

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.

G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. *ACM Trans. Program. Lang. Syst.*, 29(4):22, 2007.

A.T. Tai, K.S. Tso, L. Alkalai, S.N. Chau, and W.H. Sanders. Low-Cost Error Containment and Recovery for Onboard Guarded Software Upgrading and Beyond. *IEEE Transactions on Computers*, 51(2):121–137, 2002.

L.C. Toy. Large-Scale Real-Time Program Retrofit Methodology in AT&T 5ESS Switch. *Reliable Computer Systems: Design and Evaluation*, 1998.

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.

M. Wahler and S. Richter. Modern RTOS Features for Increasing the Availability of IEDs. Technical report, ABB Corporate Research, Research Report 2009-145, 2009.

S. Wang and K.G. Shin. Constructing Reconfigurable Software for Machine Control Systems. *IEEE Transactions on Robotics and Automation*, 18(4):475–486, 2002.