Designing for Evolvability:
Building Blocks for Evolvable Real-Time Systems

Michael Gagliardi, Ragunathan Rajkumar and Lui Sha
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
{mjg,rr,lrs}@sei.cmu.edu

Abstract

Fielded real-time systems including many defense systems, manufacturing plants and commercial aircraft avionics typically have long lifetimes ranging from a few years to even a few decades. Available technologies, system needs and customer goals change over this lifetime, and changes to a deployed system become very desirable. In this paper, we argue that such evolution must and can be supported with new system abstractions, and that real-time systems designed with these abstractions can be evolved and incrementally tested. We present two possible run-time abstractions which can act as basic building blocks to construct "evolvable real-time systems". These building blocks can be used to evolve deployed systems in general and real-time systems in particular. First, the replaceable unit abstraction allows an existing software module to be replaced online by another module with similar or enhanced functionality. Such replacement is transparent to the rest of the system. Secondly, the "cell" abstraction represents a protected module which cannot be harmed by other modules. Based on this notion is an "extensible cell", which allows a deployed module to be extended functionally without the fear of hurting its (fully certified) functionality even when the extensions can fail in unexpected ways.

These two abstractions have been implemented in a real-time POSIX testbed used in the Simplex architecture [18] and our findings are reported. Both abstractions are built on the Real-Time Publisher/Subscriber communication model [13] with modifications necessitated by safe evolutionary requirements. We conclude that guaranteed enforcement of the semantics of these two building blocks can only be provided using operating system enforced resource reservation and communication rights.

1. Introduction

The goals of this paper are two-fold. First, we present abstractions which can be used as building blocks for constructing evolvable systems. Secondly, we try to identify only those run-time, operating system (OS) and middleware mechanisms which when implemented correctly and validated, will allow systems using these abstractions to exhibit desirable properties. While these goals and the principles we apply are perhaps applicable to systems development in general, we focus only on meeting these goals in the context of real-time applications. Our treatment of this rather new perspective is informal, primarily driven by the need to design and build functioning prototypes of systems using these abstractions. In specific, we present the *software cell* and *replaceable unit* abstractions along with their associated management and communication techniques. We have successfully implemented and used these abstractions in a real-time testbed consisting of multiple feedback control devices and a real-time POSIX-compliant operating system. A formal treatment of these concepts and mechanisms is, however, expected in the future.

1.1. The Need to Evolve

Deployed real-time systems are extensively tested and must often be certified before broad-scale or commercial deployment. However, available technologies, system needs and customer goals change over time and changes to the system become very desirable. In other words, these systems exhibit the need to evolve. A fundamental bottleneck to such evolution lies in the form of required and expensive regressive/certification testing. This prohibitive cost-factor often prevents these systems from getting even relatively simple functional enhancements for fear of losing the perhaps life-critical basic functionality that has already been tested.

1.2. Designing For Evolvability

If evolutionary changes are inevitable and/or highly desired, it is only logical that a system design be made amenable to such changes. If a system can be designed such that future changes can be accommodated in a safe and incremental manner, the potential exists for a significant reduction in the cost of making future changes. We refer to such a design approach as *designing for evolvability*. The primary goals of designing for evolvability are three-fold:

- **Lower testing cost**: Given that the older system has been functioning correctly, be able to generate a smaller subset of tests to the modified system to verify its functionality without resorting to a complete battery of regression and certification tests as if the entire system were new.
- **Safety**: Any new changes, if erroneous, do not compromise the older basic functionality. An acceptable alternative may be to at least contain the damage that can occur due to problems in the new changes.
- **Extensibility**: New changes and/or enhancements must be easier to introduce.

1.3. Related Work

Type-checking in modern programming languages such as Ada 83 [1], Ada 95 [2] and C++ is intended to achieve effects similar to what we attempt to achieve albeit at a lower level of abstraction within a process. Our approach differs in three significant ways. One, our focus is relatively coarse-grained compared to fine-grained type-checking of variables. Two, our focus is not on compile-time checks but on system mechanisms including run-time primitives and middleware abstractions. Finally, we consider how existing functionality within a process can be safely extended by adding new processes.

Programming languages such as Modula III have introduced concepts such as safe and unsafe modules in order to derive safety properties when modules interact. In the SPIN operating system [4], kernel extensions written in an extended Modula III language can be dynamically linked into the kernel while guaranteeing that these extensions cannot corrupt other core kernel services. Again, the focus here is on compile-time capabilities of the language and therefore require that the kernel extensions be written using the language. In contrast, we seek language-independent mechanisms and abstractions for safe extensions. The work on SPIN also does not address real-time concerns.

Our requirements and goals lead us to the conclusion that mechanisms such as address space protection in a multitasking environment, resource reservation on the CPU as supported by RT-Mach [10] and/or fixed priority scheduling [9], bandwidth reservation on the network as available in ATM networks, and rights-based communication capabilities as supported in Mach [14] and RT-Mach are critical components for building evolvable (real-time) systems.

1.4. Brief Overview of The Real-Time Publisher/Subscriber IPC Model

A brief overview of the real-time publisher/subscriber (RT P/S) interprocess communication model [13] is given below since its capabilities are put to significant use in this paper. The RT/PS allow processes running on different machines to publish and receive real-time data in a location-independent and protocol-transparent way. Specifically, a consumer of a data item need not know the producer of that data item, and vice-versa. In addition, the communication protocol used between the data producers and data consumers is transparent to the application.

Publishers act as data sources and subscribers act as data sinks. A publisher can also be a subscriber and vice-versa. A *distribution tag* (or *name*) identifies each category of data item published. Many publishers can publish on the same tag, and many subscribers can receive on the same tag. The publisher on a tag need not know how many subscribers are subscribed to that tag. Similarly, a subscriber to a tag need not know how many publishers are publishing on a tag. All that the application(s) have to do for using this communication model is to agree upon tags (names) of the data items that will be published and their data formats.

The Publisher/Subscriber Inter-Process Communication Model is a variant of group-based communications [6]. The real-time version of the abstraction addresses not only issues of programming ease, portability and scalability but also analyzability for real-time systems. The model has been used successfully in building a software architecture for building upgradable real-time systems.

2. The Replaceable Unit Abstraction

In this section, we present the "replaceable unit" abstraction and our realization of the abstraction using a replacement manager and a uniform communication model. We then discuss how a unit can be replaced online and our experience with the abstraction in a real-time application.

2.1. Defining and Understanding the Replaceable Unit

The `replaceable unit` is a software abstraction which allows even a real-time software module to be replaced on the fly while the system continues to function normally.

The basic idea behind the replaceable unit abstraction is as follows. A module interfaces with the rest of the system by sharing or communicating some data. Suppose this interfacing is made to use logical data sharing and communication endpoints instead of endpoints hardwired to the module in question. Let us assume that the replacement of the module is then needed, to add to or to fine-tune existing functionality for evolution purposes. The mapping of the logical endpoints can be dynamically modified to also map to the new module, and when the new mappings are in place, the mappings to the old module can be turned off. As far as the rest of the (older) system is concerned, the new module has replaced the old module with the exact same connectivity. The real-time publisher/subscriber inter-process communication model which provides transparent communications between sender(s) and receiver(s) is a good choice for realizing the basic idea of dynamically mapping communication endpoints.

The replacement operations must be coordinated such that they get initiated, progress and complete correctly (hopefully) without disrupting the continuing functionality of the system. There are two ways of managing replacement operations:

1. **Supervisor-Triggered**: The replacement of a software module is triggered and managed by an external "replacement manager".
2. **Self-Triggered**: The replacement of a software module is triggered and managed by the module itself.

In the rest of the section, we shall focus mostly on supervisor-triggered replacement operations, since much of this discussion is applicable to self-triggered replacement operations as well.

2.2. Realizing Replaceable Units

Our realization of the replaceable unit encapsulates a process with a well-defined communication template. Replaceable units encapsulate a process in order to take advantage of the address space protection and concurrency available to standard processes. In this way, multiple replaceable units can co-reside on a processor and are protected from accessing other replaceable units' address spaces.

Replaceable units are created and managed by replacement managers. A complete hierarchy of replaceable units and replacement managers can be sustained by the abstractions provided. That is, replacement managers can manage a set of replaceable units, who in turn can also manage another level of replaceable units and replacement managers, and so on. The replaceable unit management functionality has been captured in a uniform application programming interface (API). Replaceable units also communicate with each other via a uniform communication API.

Each created replaceable unit contains a thread which establishes bi-directional communication with its replacement manager, namely the replaceable unit which created this replaceable unit. The nature of the communication is to enable the replacement manager to perform certain operations on their subordinate replaceable units by means of communicating between the replacement manager and the replaceable units' threads. The methods supported between replacement managers and their replaceable units' threads is discussed next.

2.3. The Replaceable Unit Communication Model

The replaceable unit communication model is built on top of the Real-Time Publisher/Subscriber interprocess communication model with some added capabilities. A replaceable unit communicates with other replaceable units via this interface.

In order for replaceable units to properly support online replacement, the replacement manager must retain information about the logical communication channels in use, namely the distribution tags. Data sources always publish on the same logical tag(s), while data sinks always subscribe to the same logical tag(s). The receivers (senders) on a tag can change dynamically but the senders (receivers) continue to communicate using the same tag(s).

Each time a replaceable unit performs an IPC operation, with the exception of "send" and "receive", the replacement communication thread notifies its replacement manager of the operation. The replacement manager can then control the timing of the *enabling* and *disabling* of the I/O operations of the replaceable units it is managing. This separates the concerns of switching between replaceable units from the replaceable units themselves. As a result, the developers of a replaceable unit can concentrate on the application details without being concerned about the timing and mechanics of the on-line replacement activities.

2.4. Replaceable Unit Management

A *replacement transaction* is used to replace online one replaceable unit with another. During this transaction, state information may need to be transferred from the original unit to the new replaceable unit. Alternatively, the new unit may need to capture the dynamic state information of the original unit to help avoid any undesirable transients in the behavior of the new replaceable unit when it comes online. To enable smooth "switching" between new and old components, the sending and reception on their tags are enabled and disabled selectively. This sequence is illustrated in Figure 2-1 when a replacement manager is used.

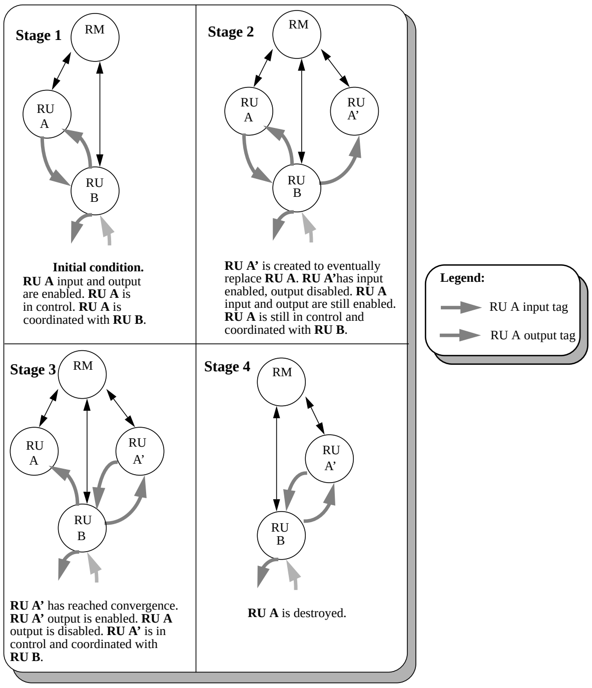

Figure 2-1: The Conceptual Sequence for Supervisor-Triggered RU Replacement

2.5. Replacement Steps Using a Replacement Manager

The replaceable transaction of a single replaceable unit is summarized in the following stages:

1. The new replaceable unit is created.
2. New input and any state information is provided to the new replaceable unit when it is ready. The new unit begins computations based on the input data, however, its output is disabled.
3. The replacement manager will monitor the output of the new unit to synchronize or converge with respect to the rest of the system.
4. Upon convergence or synchronization, the replacement manager will then disable the output of the old unit and enable the output of the new unit, thus switching control from the old unit to the new unit.
5. The replacement manager may then delete the old unit.

It must be noted that a replacement manager can manage any number of replaceable units.

It must be noted that some parallels can be drawn between the replaceable unit abstraction and techniques for building highly available services using either a primary-backup approach [5] or a state-machine approach [15]. However, there are three primary differences. One, we treat all modules as potential replaceable units instead of treating only servers as being dynamically created to provide continuous service. Two, we exploit the nature of real-time systems while bringing in a new version of a replaceable unit: only recent real-time data needs to be transferred (along with some long-lived data if needed). Thirdly, the real-time replacement of a replaceable unit must be accomplished without endangering real-time constraints of the system.

2.6. Replaceable Units in a Real-Time Application

Our replaceable unit abstraction has been successfully implemented in C++ and fully tested in a real-time POSIX environment [11]. We have tested the abstraction in the following existing application framework for purposes of validating the replaceable unit abstraction which can be used in various ways. The application testbed which we use employs an inverted pendulum device which is unstable outside of the control loop. The inverted pendulum consists of a motorized cart which rides on a straight track and supports a freely-hinged rod which is to be balanced by sensing the rod angle and properly moving the cart to keep the rod from falling. The objective of a feedback controller is to balance the rod and keep the cart near the center of the track. COTS software and hardware are used as much as possible in the testbed which includes a POSIX-compliant real-time operating system on an Intel x86 platform.

The application software was partitioned into replaceable units and a replacement manager. The replacement manager also acts as a user interface to control the demonstration. The demonstration is interactive in that users can create a new controller *on the fly* and replace the existing baseline controller. As these new controllers enter and leave the feedback loop, continuous control of the inverted pendulum is maintained. These controllers are in fact replaceable units which conform to the replaceable unit abstraction discussed above and adhere to the replaceable unit communication protocol.

The users can make any application change they wish by editing a controller source file, compiling, and linking the new replaceable unit while the baseline controller is controlling the device. Once the program is successfully linked, the user can instruct the replacement manager to introduce this new controller into the job mix and take control of the inverted pendulum device. Per the Simplex architecture being developed by the Software Engineering Institute [18], if the new controller does not control the inverted pendulum to the required degree of performance, the new controller will be automatically disabled and the old controller will gain control of the device. In this way, the system will never be worse off than the previously established baseline controller. If the new controller does meet the required degree of performance, the new controller will remain in control. This demonstration utilizes a single processor and can tolerate only application level software faults. We also maintain a testbed which contains a triplicated PC architecture which tolerates not only application level software faults but also tolerates one active *and* one passive hardware or operating system fault.

3. The Cell Abstraction

In this section, we present a "cell" abstraction which can be used to isolate trusted modules from untrusted modules. Specifically, cells can allow a trusted module to continue to function correctly even under the co-existence of other faulty modules. We first define some basic terms, and then discuss how this abstraction can be supported by underlying run-time systems.

3.1. Basic Definitions

We define some some basic terms below.

- A `cell` is a module which cannot be corrupted by other modules in a computing system.
- A `software cell` is a software module which cannot be corrupted by other software modules in a computing system.
- A cell is said to be `extensible` if it can export its internal data to other modules without being corrupted.
- A `real-time cell` is a cell whose timing constraints cannot be violated, and its logical correctness also cannot be corrupted due to other modules.
- A `trusted cell` is a cell whose functionality is known to be correct (or acceptable).

The above terms may be combined. For example, a real-time software cell can also be made extensible and trusted.

The motivation behind the notion of a trusted cell is that when a system has been functioning for a while, trust in it builds. As a result, when incremental or evolutionary enhancements need to be made, one would like an existing system module be treated as a trusted cell. Its functionality must not be impaired by new modules or changes in other modules.

3.2. A Cell As a System-Level and/or Software Abstraction

It is important to note that the definitions of a cell and a real-time cell above do not specifically refer to a hardware module or a software module. This distinction is important since the simplest implementation of a cell is to physically isolate a trusted module from other untrusted modules. This scheme is often carried out in mission-critical and safety-critical application such as nuclear power plant control, avionics in commercial flights, etc. where physically isolated nodes execute simpler code to ensure that the safety of the system is not violated under any circumstances.

The `cell` abstraction is actually motivated by this physical separation of modules which acts as a clean and easily provable barrier between mutually untrusted computing nodes. The `software cell` is intended to achieve the same mutual protection that physically isolated modules guarantee. If such protection can be provided, both trusted and untrusted modules can be allowed to co-exist on the same node by encapsulating software modules as software cells.

The realization of a cell as a physically isolated system module is not very useful in many circumstances. Each time new functionality is added, it is often impractical to assume that they be added in physically separate modules with no connectivity to the older system. For example, the FAA's Traffic Alert and Collision Avoidance System (TCAS) [19] has a requirement that an audible alarm be sounded if two or more aircraft are close to one another during flight. Suppose that such a TCAS system needs to be extended to add friendlier or more sophisticated functionality. Due to space constraints and requirements for access to older data, it is generally desirable to accommodate the changes in software on the same processor(s). The extensible cell is specifically defined for this purpose so that a module can be extended in a trustworthy manner.

Since the physical isolation between system modules is an obvious (but not always useful/meaningful) way to extend or evolve functionality, we shall focus only on the software aspects of cells where two or more cells co-exist in a physically non-isolated environment. Hence, we shall use the terms `cell` and `software cell` interchangeably. We shall also restrict our attention to a uniprocessor environment; we shall touch upon implications to distributed systems in Section 5.

3.3. Module Requirements that Satisfy Software Cells

A software cell is not corruptible by other software modules. A software module needs to use a whole collection of resources including, but not limited to, some portion of the CPU cycles, its code segment, its data segment, its stack segment, the devices it accesses, the files it needs, etc. We now informally state some properties that cells exhibit. Our current focus is to define sufficient but simple conditions for software modules to satisfy these properties. Future work will address necessary conditions and formal proofs for these properties.

- **Exclusive Access Property**: A software module all of whose run-time resources are dedicated to and accessible only by the module is a software cell.

  This property follows from the fact that if all run-time resources are dedicated to and accessible only by the software module, it is equivalent to achieving physical isolation. However, one must be very careful about how shared resources such as the CPU are allocated/accessed.

- **Exclusive And Immediate Write Access Property**: A software module all of whose run-time resources are dedicated to, and modifiable only by the module whenever it needs to, is a software cell.

  This property follows from the fact that if the module can write to all resources it needs when it needs to and no other modules can modify those resources in any way, then it is equivalent to having the resources exclusively accessible by the module. The previous property then applies.

- **Trusted Communications Property**: A correctly functioning cell that communicates only with trusted cells can be trusted.

  The above property merely skims the various aggregation and composition properties that arise when one or more cells need to communicate. If *all* the senders and receivers of messages to a real-time module are trusted real-time cells, then the real-time module is a real-time cell when it is indeed a cell in the absence of any such messages. However, since this property is recursive, no cell in the communication chain can be untrusted.[1]

  Properties less constraining than the above are possible in the context of more constrained communications and can be exploited to construct `extensible cells`. Two of these properties are discussed below.

- **Blind Sends Property**: A module whose only external interaction is to send messages to other modules and which does not depend on messages from other modules for its correct operation is a cell.

- **Blind but Limited Sends Property**: A real-time module whose only external interaction is to send messages to other modules with a known maximum time penalty and without depending on any receptions for its correct operation is a real-time cell.

In the last two cases, the cell spits out some data but does not depend on the results of the sending for its own correct operation. In other words, the sending is purely for the benefit of the recipients (if any). In the last case, the real-time cell can only have a predetermined worst-case timing penalty from sending out messages to ensure its own timing correctness.

3.4. Some Applicable Scheduling and Run-Time Notions

Priority-driven real-time systems either use fixed priorities [9, 8] or dynamic priorities [9, 3, 7]. However, it is often noted that, when a temporal overload occurs, the fixed priority scheme can be used as is or after period transformation to ensure that the most critical tasks can still meet their deadlines [16]. This property of fixed priority scheduling is not useful for avoiding transient overloads alone. In fixed priority scheduling, *all* instances of a task with a higher priority task take *strict* precedence over *all* instances of lower priority tasks. In other words, if instances of two tasks with different priorities are eligible to run, an instance of the higher priority task always takes precedence. We refer to this property where a task instance

---

[1] This property is conservative, but not as conservative as it may first appear. Consider, for example, security needs in institutional gateways into the insecure internet. If communications only with trusted Internet counterparts is allowed, the functionality offered to (say) World Wide Web surfers and users within the institution does become seriously restricted. If this restriction is lifted, however, 100% security is not guaranteed. However, attacks on internal data servers and mainframes can still be detected and stopped.

can always force preemption of another task instance predictably and enforceably as **strict precedence**.[2] While this is an obvious property, as we shall see, it has important implications to the implementation of cells.

Similarly, if instances of two or more tasks are waiting for the use of the same shared resource, the instance of the highest priority task must take precedence in using the resource. This notion is the basic driving concept behind the family of priority inheritance protocols [12, 17]. With one of these (fixed priority-based) protocols, the longest amount of time that an instance of a task is blocked by instances of lower priority tasks can be bounded. And once the resource is released, the instance of the highest priority task that is waiting for the resource takes precedence in using it. We refer to this property of fixed priority and priority inheritance protocols as **bounded-wait precedence**.

In the context of cells and evolvability, one must re-interpret these notions. Suppose that the instance of a lower priority task locks a shared resource, but fails while inside the critical section. If the run-time system cannot automatically release the lock (and clean up the critical section if applicable), the system is said *not* to exhibit the "bounded-wait precedence" property.

Start-up precedence control and **controlled access to resources** are two other notions that are also very helpful in constructing cells. The former property requires that on system startup (booting up), `trusted cells` be started and run ahead of the other modules. The latter property requires that the number of modules which can access a shared resource be controllable. For example, by using startup precedence control, a trusted cell can start first, limit the number of accessing modules to its needed resources to be 1 and obtain those resources. Any subsequent attempts by other trusted cells to obtain these same resources will therefore fail. For example, if a trusted cell needs access to a special device file, and at most one process can open and use the file at any given time, the file is said to have the controlled access property.

3.5. Implications to Real-Time Systems

As mentioned earlier, our goal is to identify only those run-time, OS and middleware mechanisms which when implemented correctly and validated, will allow one to implement evolvable real-time systems. In this section, we try to identify techniques by which `real-time cells` can be built with both existing and newer mechanisms. When limitations of existing systems stand in the way, we point out guidelines that can be used while inserting changes to isolate and identify possible introduction of problems into the system.

It is sufficient to use enforced reservation mechanisms to satisfy the Exclusive and Immediate Write Access property. Examples of such reservation mechanisms are the bandwidth reservation model of ATM and the resource reservation model used in the Real-Time Mach operating system. As a result, the Exclusive and Immediate Write Access property and the notion of a real-time cell have many interesting implications for real-time software modules.

1. **Temporal Protection of CPU Resource Requirement**: First of all, consider a software module which must *only* perform computations (say) in a periodic fashion. If the module is encapsulated into a process with address space protection, intrusion from other processes cannot occur. Similarly, if its CPU requirements can be guaranteed by the underlying run-time system, then the software module can be treated as a *real-time cell*. In practice, the processor reservation model in Real-Time Mach [10] which can guarantee that an application will obtain C units every T units of time provides exactly these semantics.

   In the absence of such CPU resource reservation, cells can still be implemented by requiring that all *trusted real-time cells* have higher priority than other cells and exploiting the strict precedence property of fixed priority scheduling. In other words, the enforcement mechanism (guideline) must lie with the developers and the testers. Any changes in priority are potential threats and must be checked, validated and certified.

2. **Input/Output**: Consider a real-time software module which must perform I/O (say with the physical environment to perform feedback control of a physical device). As per the Exclusive and Immediate Write Access property, this module can be treated as a `real-time cell` if these devices can be dedicated or guaranteed to the cell requiring them. This requirement can again be satisfied in one of two ways:

   - by reserving the I/O devices for guaranteed use by the module, or by
   - supporting startup precedence control and exclusive access to the devices. If the module consisted of one process alone, the access can be limited to one caller. For example, in a real-time POSIX environment, devices may be represented as special files, and the drivers for these devices can allow only the first "open" operation to succeed and let successive attempts fail. Similarly, in an RT-Mach environment, send rights to an unused device port must be made available only to the first requester. If the cell consisted of more than one process, the maximum number of accesses must be set accordingly.

3. **Resource Sharing**: Consider a real-time software module which needs access to a shared resource. One may be tempted to just

[2] "Strict precedence" can also be achieved with a dynamic scheduling policy like the earliest-deadline-first policy. However, a reservation-based approach where a task which exceeds its stated computational requirements can be forcefully preempted despite its earliest deadline becomes necessary.

apply startup precedence control and controlled access to these resources as well. This is certainly possible under several circumstances such as in the use of memory buffers, pages and communication buffers.

Consider a real-time module which accesses a shared resource in write mode. If the resource is read-locked from another module which then fails, the real-time module will be prevented from getting its write-lock. Converting all locks to write-locks does not address the problem either because of the possibility of failure. A clean solution to this problem remains elusive and entitles further study. The `extensible cell` that we shall discuss shortly addresses this problem only by making all resources used by a cell local and exporting any related data in a selective fashion.

4. **Communications**: Communication requirements are perhaps the hardest to deal with in the context of cells. Since a `real-time cell` must not be corruptible by other modules, its communication ports must be unreachable by these modules. If other modules can send fake messages or just unexpected messages to a cell, the cell must look at these messages and even if it dropped them, its timing behavior can be altered and seriously compromised.

   One also needs to be concerned about communication end-points used for sending messages and those used for receiving messages. To maintain timing and semantic guarantees, a communication port used by a `real-time cell` for sending must be useful for sending only by the cell, and a communication port used for receiving must receive messages only from the cells' intended recipients. These requirements are almost restatements of capability-based communications, where one must have a capability/right before sending from or to a port. It is noteworthy that the Mach and RT-Mach communications mechanisms are based on ports and port-rights. Rights to send to a port and receive on a port must be explicitly obtained by a process before these operations can succeed. Furthermore, these rights cannot be counterfeited. Furthermore, with support for strict precedence, a trusted cell can preempt an untrusted cell using a higher priority and make use of its resources.

   We consider two common communication end-points namely sockets and POSIX message queues. At the sending side, once a process binds a socket to a particular port number, no other process can obtain that port number. With startup sequence control, this adequately solves the problems for cells which act as senders. However, any process can send messages to a socket at a port number. Hence, a cell which needs to receive messages at a socket port can be overwhelmed by an errant and/or untrusted module.

   In contrast, the rights to use POSIX message queues are controlled by classical Unix file system privileges, namely different OS-enforceable privileges for the user/owner, group and all others. Hence, a reasonable way of using POSIX message queues for constructing cells is to assign communicating cells a different group-id and to assign read/write privileges only for appropriate group members. The granularity and the management of user groups need to be closely checked when any changes or additions are made.

In summary, the following requirements are needed for constructing real-time cells:

- **Address space protection**: a cell must reside in its own address space(s) such that its code, data and stack segments cannot be modified by other software modules.
- **CPU guarantees**: the CPU bandwidth for a real-time cell must be reserved and guaranteed. With these guarantees, a trusted cell does not depend upon the trustworthiness of other modules. If such guarantees are not available, startup precedence control must be used to start all *trusted* cells first and they must be assigned higher fixed priorities than other software modules. Any errors in a higher priority trusted module can possibly affect the timing behavior of another trusted module.
- **Resource guarantees**: any resource that is accessed by the real-time cell must be dedicated/guaranteed to the cell. The CPU guarantee can indeed be considered to be a special case of this requirement except that the CPU is scheduled much more dynamically and flexibly than other typical resources.
- **Trusted communications**: A real-time cell can spend a known amount of time in sending out messages. Our real-time cell currently can accept messages only from cells with trusted communications.[3]

4. Extensible Cells with The Real-Time Publisher/Subscriber IPC Model

In the previous section, we studied how the correctness properties of cells can be preserved such that they cannot be damaged by other modules. However, the functionality of an evolvable real-time system often needs to be extended without endangering existing functionality. An `extensible cell` allows such safe extensions to existing functionality. In this section, we present the basic concepts behind an extensible cell, and discuss its design and

[3]Relaxations of this constraint are possible but are beyond the scope of this paper.

implementation in a real-time POSIX environment using the real-time publisher/subscriber communication model.

One can exploit the blind but limited sends property to extend the functionality of an existing cell. Two basic ideas underlie our extensible cell:

1. **Data Export**: A cell can *export* data that is internal to it. Data, thus exported, can be used by other software modules as they see fit. This contributes to extensibility of functionality.
2. **Safe Export**: Any data export by a cell must occur in such a way that the cells logical and timing behavior is not modified. This contributes to the safety of the cell exporting the data.

In order to test the requirements that we impose on cells and extensible cells, we made the necessary changes to a POSIX-compliant real-time operating system and experimented with a real-time feedback control system as an application that needs to be safely extended.

4.1. The Application Testbed

The real-time feedback control system we use is an *inverted pendulum* discussed in Section 2.6 which consists of a freely hinged rod, mounted on a cart which can be moved along a uni-dimensional track. The position of the cart is controlled by applying an appropriate voltage to a motor which can move the cart along its track. The angular position of the rod and the position of the cart act as inputs to the computing system performing the feedback control operations. We have had considerable experience with this feedback control system for almost 2 years now that we have a very high degree of confidence in the feedback control algorithm.

The new system functionality that we specified for this experiment was that the inverted pendulum be drawn continually in a X11-window to provide a real-time animated graphical view of the pendulum. Since the X11 window can be displayed on a remote screen, this would allow local or "tele-viewing" of the pendulum under control. The primary requirement was that we should use the notions of a cell and an extensible cell to build this functional extension in a safe manner such that the feedback control of the pendulum itself would not be affected.

4.2. The Inter-Process Communication Model

The real-time publisher/subscriber communication model is an ideal match for supporting our notion of extensible cells. Under this IPC model, a sender can publish messages without any knowledge of who the receivers (subscribers) are, and an extensible cell has no way of knowing who the receivers of its information will be since these receivers may only be created well in the future as the system needs to evolve.

Based on the principles and requirements described in the previous section in general and Section 3.5 in particular, changes needed to be made to the overly flexible design and implementation described in [13]. In particular, to satisfy the requirements of the Blind but Limited Send Property, we needed to extend the attributes of a distribution tag and modify the distribution tag programming interface so as to have enforceable and bounded timing penalty characteristics for the sender of a message. We added the following enforceable features to a distribution tag:

- Only the creator of a tag can destroy a tag. This disallows errant modules from deleting the tag(s) used by a real-time cell.
- The creator of the tag can specify the maximum number of creators (`CreateDistributionTag` calls) allowed on the tag. By choosing this value to be 1, a cell can ensure that no other module can destroy its tag(s). Instead, a module which needs a handle to an existing tag can call a new API function `GetDistributionTag` where formerly it would call `CreateDistributionTag`.
- The creator of the tag can specify the maximum number of publishers on the tag. By specifying this value to be (say) 1, a cell can ensure that it is the only publisher of valid messages on that tag.
- The creator of a tag can specify the maximum number of subscribers on the tag. Message publication is carried out in the execution of the calling thread to keep the steady-state execution path for sending messages very small. By specifying the maximum number of subscribers to be (say) 1, a cell can ensure that it does at most the work corresponding to sending one message and no more.

By using startup sequence control, a trusted cell can start first and create the tags it needs with the attributes it desires - typically 1 creator and 1 publisher (namely itself) and 1 subscriber.

Note: Even the above enhanced features offer guaranteed enforcement semantics only if all the applications perform inter-process communications by going through the Publisher/Subscriber interface. This is because the underlying communication primitives including sockets and POSIX message queues do not have enforced send-receive rights or are at too coarse a level.

4.3. Device I/O Support

Feedback control of physical devices need to perform analog I/O. To enforce *real-time cell* semantics for control of the inverted pendulum, we modified the analog I/O kernel device drivers such that at most one process can open and access them at any given time. Again, by start sequence control, the pendulum control process is started first and obtains access to the I/O ports.[4] Device accesses are enforced by the kernel driver and offer strong semantics.

[4]It may be tempting to let other processes at least have read accesses to the input ports so that they can sample the state of the controlled plant. Unfortunately, our testbed experience has been that very frequent reads disrupt the signal-to-noise ratio of the input signals due to capacitive effects.

4.4. Exporting Data from the Extensible Cell

The intent of an extensible cell is to export internal data for possible extension by future modules so that the system can evolve without endangering existing functionality. In order to facilitate this process, we use traditional header file formats to export the format(s) of the data that an extensible cell exports. This header file is referred to a *data export file* and includes the distribution tag(s) that the data is exported on, and the format of the data publication.

One (or more) extension modules can import this export file (by including the header file) and use the tag information and data format appropriately. The `animate_pendulum` extension module subscribes to the information published by the `control_pendulum` extensible real-time cell, and then based on the values that it receives, it provides an animated real-time view of the pendulum. Note that the displayed data uses filtered values computed *internal* to the `control_pendulum` cell, providing not only a better view of the system state but also a safe way of extending the feedback control algorithm.

4.5. Limitations

Our above implementation of extensible cells has a rather serious limitation. Future extensions only have access to data that is currently published by old trusted modules. If a future extension needs some other data, problems begin to crop up. We are therefore exploring alternative and more flexible ways of implementing extensible cells.

4.6. Cells as Replaceable Units or Vice-Versa

Two natural questions to ask are whether a cell can be structured as a replaceable unit, and whether a replaceable unit can be structured as a cell. The conceptual answer is in the affirmative but a faithful implementation of their semantics depends on the communication capabilities (or lack thereof) in the underlying run-time system. As discussed in Section 2, inter-process communications are fundamental to the notion of a replaceable unit which can be modified/replaced at run-time.

- **Source/Destination Identification**: If the sender or the receiver of a message cannot be authenticated, then guaranteed semantics with respect to untrusted modules are at best murky. An errant/untrusted module can act like an authorized cell, and force unintended actions.
- **Dynamic Operations**: Startup sequence control also ceases to be a good solution since the decision to bring in a component is dynamic and typically happens well after other untrusted modules have been running for a while.

Suppose that the kernel can dynamically enforce send and receive rights, that sender and receiver id's cannot be faked, and that these rights can also be selectively passed around by application processes. Then, both the above problems can be addressed effectively.

5. Concluding Remarks

Deployed real-time systems get completely replaced only on an infrequent basis. Most systems continue to evolve over time to suit changing requirements. However, due to the ever-lurking risk of losing or damaging existing functionality, changes are often slow and expensive. We have therefore argued that systems must be built with an eye towards being amenable to future extensions and modifications. We refer to this as "designing for evolvability". We present two possible abstractions, a replaceable unit and a cell, which have been specifically designed to meet this goal. A replaceable cell can be used to replace a software module (online or offline), and a cell is immune to changes outside itself. New systems designed with these abstractions have built-in mechanisms for evolution and can be extended in a dynamic and/or safe way. We find that the semantics of these abstractions can be strictly enforced only with kernel/OS support. However, even the application of guidelines (such as using only the real-time publisher/subscriber model for IPC) can significantly benefit a system's evolutionary aspects by narrowing down the number of new tests that are needed.

Our perspective towards providing system abstractions to build evolvable real-time systems seems rather new. However, this paper only takes a relatively informal look at the abstractions, concepts and mechanisms needed to build such systems. We expect that future work in this domain will include formal notation, theorems and proofs. In particular, such notation must consider the interactions between different kinds of modules and cells, and what properties hold good under different kinds of cell aggregations and compositions. It must also be noted that the properties we state are valid only for uniprocessor or centralized systems, with some problems yet to be solved in this context. Remote authentication and traditional security issues become critical concerns in the context of distributed real-time systems. If the underlying network infrastructure is not secure or does not support reserved bandwidth communications, then hard guarantees of semantics may be unlikely.

References

1. *Reference Manual for the Ada Programming Language.* U.S. Department of Defense, Washington, D.C., 1983.

2. *Ada 95 Reference Manual.* International Standard, ANSI/ISO/IEC-8652:1995, January 1995.

3. Baker, T. "Stack-Based Scheduling of Realtime Processes". *Journal of Real-Time Systems 3*, 1 (March 1991), 67--100.

4. B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, C. Chambers, S. Eggers. "Extensibility, Safety and Performance in the SPIN Operating System". *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Dec 1995).

5. N. Budhiraja and K. Marzullo and F. B. Schneider and S. Toueg. Primary-backup protocols: Lower bounds and optimal implementations. Tech. Rept. TR 92-1265, Department of Computer Science, Cornell University, January, 1992.

6. Ken Birman and Keith Marzullo. "The ISIS Distributed Programming Toolkit and The Meta Distributed Operating System.". *SunTechnology 2*, 1 (1989), .

7. Jeffay, K. Scheduling Sporadic Tasks with Shared Resources in Hard Real-Time Systems. TR90-038, Department of Computer Science, University of North Carolina at Chapel Hill, November, 1989.

8. Klein, M. H., Ralya, T., Pollak, B., Obenza, R. and Harbour, M. G. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems.* Kluwer Academic Publishers, 1993. ISBN 0-7923-9361-9.

9. Liu, C. L. and Layland J. W. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment". *JACM 20 (1)* (1973), 46 - 61.

10. C. W. Mercer and R. Rajkumar and J. Zelenka. Temporal Protection in Real-Time Operating Systems. Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software, May, 1994, pp. 79-83.

11. *IEEE Standard P1003.1b (Realtime Extensions to POSIX).* IEEE, 345 East 47th St., New York, NY 10017, 1993.

12. Rajkumar, R. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, 1991. ISBN 0-7923-9211-6.

13. Rajkumar, R., Gagliardi, M. and Sha, L. "The Real-Time Publisher/Subscriber Communication for Inter-Process Communication in Distributed Real-Time Systems". *The First IEEE Real-time Technology and Applications Symposium* (May 1995).

14. R. Rashid and R. Baron and A. Forin and D. Golub and M.Jones and D. Orr and R. Sanzi. Mach: A Foundation for Open Systems. Proceedings ofthe Second Workshop on Workstation Operating Systems (WWOS-II), Sept., 1989, pp. 109-113.

15. F. B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". *ACM Computing Surveys 22*, 4 (December 1990).

16. Sha, L., Lehoczky, J. P. and Rajkumar, R. "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling". *IEEE Real-Time Systems Symposium* (1986).

17. Sha, L., Rajkumar, R., Son, S. H., and Y. Lee. "A Real-Time Locking Protocol". *IEEE Transactions on Computers 40*, 7 (July 1991).

18. Sha, L., Rajkumar, R. and Gagliardi, M. "The Simplex Architecture:An Approach to Build Evolving Industrial Computing Systems". *The Proceedings of the IPC, International Automation Week* (1995).

19. *Introduction to TCAS-II.* Federal Aviation Administration, U.S. Department of Transportation, Washington, D.C., 1990.

Table of Contents

List of Figures