Using Containers to Enforce Smart Constraints for Performance in Industrial Systems

Scott A. Hissam
Gabriel A. Moreno
Kurt C. Wallnau

August 2005

Predictable Assembly from Certifiable Components

Technical Note
CMU/SEI-2005-TN-040

This work is sponsored by the U.S. Department of Defense.

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Contents

List of Figures

List of Tables

Abstract

Today, software engineering is concerned less with individual programs than with large-scale networks of interacting programs. For large-scale networks, engineering problems emerge that go well beyond functional correctness (the purview of programming) and encompass equally crucial nonfunctional qualities such as security, performance, availability, and fault tolerance. A pivotal challenge, then, is to provide techniques to routinely construct systems that have predictable nonfunctional quality. These techniques impose constraints on the problem being solved and on the form solutions can take. This technical note shows how smart constraints can be embedded in software infrastructure, so that systems conforming to those constraints are predictable by construction.

1 Introduction

Companies rely on software to deliver innovative solutions to customers. Such software often executes in environments that have strict timing, safety, reliability, and security requirements. Software malfunctions in these applications are expensive and possibly catastrophic. The steep cost of developing highly reliable software poses a challenge for the entire software industry. The scale of today's systems, not to mention that of tomorrow's, exposes the fundamental inadequacies of relying on testing to achieve high assurance. The Predictable Assembly from Certifiable Components (PACC) Initiative at the Carnegie Mellon[®] Software Engineering Institute (SEI[SM]) has developed an approach to ensure that the critical runtime behavior of systems is predictable by construction, to reduce testing costs and hasten the introduction of new high assurance software into the market.

The goal of the PACC Initiative at the SEI is to enable the construction of software systems from components in a manner that allows for automatic prediction of system behavior [Wallnau 03a]. This goal is realized by developing or enhancing component technologies, using and extending property theories, and developing prototype tools and methods. This technical note focuses on an approach to enhancing a component technology as a means to ensure that software systems are predictable by construction.

1.1 Smart Constraints

Constraints lie at the heart of all engineering disciplines. An engineering problem may present unique challenges, but the skilled engineer knows how to coerce it into a form that can be solved with proven and well-defined techniques. These techniques impose constraints on the problem being solved and the form solutions can take. Making it possible to solve entire classes of problems predictably and routinely more than compensates for the loss of freedom implied by these constraints.

Today's software engineering is concerned less with programs per se than with large-scale networks of interacting programs. For large-scale networks, engineering challenges emerge that go well beyond functional correctness (the purview of programming) and encompass equally crucial nonfunctional qualities (sometimes called quality attributes [Boehm 78]) such as security, performance, availability, and fault tolerance. A pivotal challenge for software engineering research is to provide techniques to routinely construct systems that have predictable nonfunctional quality. It follows, from our earlier assertions about engineering

[®] Carnegie Mellon is registered in the U.S. Patent and Trademark office by Carnegie Mellon University.

[SM] SEI is a service mark of Carnegie Mellon University.

disciplines, that these techniques will likely impose constraints on how future software systems will be constructed.

This technical note shows how “smart” constraints can be embedded in software infrastructure so that systems conforming to those constraints are predictable by construction.

1.2 About This Note

Section 2 establishes the container idiom as a method for introducing smart constraints and explains how it is important for making predictions. Section 3 discusses the need for prediction in an industrial setting and describes the application of the PACC performance reasoning framework [Hissam 04b] to a problem in that setting. Section 4 illustrates how the Pin component technology[1] was extended to use the container idiom to enforce the constraints assumed by the PACC performance reasoning framework. We summarize in Section 5.

[1] Pin is a basic, simple component technology suitable for building embedded software applications [Hissam 05].

2 The Container Idiom

The approach we use is governed by two premises: (1) smart constraints can be defined that lead to systems with predictable runtime qualities and (2) component technology can be made to package and enforce constraints to make software predictable by construction.

These premises are supported by two key assertions: (1) a runtime quality must be defined in terms of observations that can be made on execution traces, and (2) a runtime quality is predictable if and only if there is a theory for predicting future observations. The crucial points here are that quality is defined relative to a predictive theory and that this theory must yield confidence about its predictions.

The value of predictive theory is not new in science or in software engineering. The timing behavior of a software system may be predictable using generalized rate monotonic scheduling theory [Klein 93] or real-time queuing theory [Lehoczky 96]. Both theories (generally all theories) make assumptions about the systems that are their subjects, and any system that satisfies these assumptions is predictable. Smart constraints ensure that the assumptions are satisfied; the constraints are smart because they are informed by predictive theories.

It is one thing to define a smart constraint, but it is another to guarantee (rather than assume) the constraint is satisfied. One recurring component technology idiom is depicted in Figure 1. This idiom is particularly effective in packaging smart constraints because it imposes strict rules on visibility, coordination, and other runtime behaviors. These, in turn, provide “hooks” on which to hang additional smart constraints.

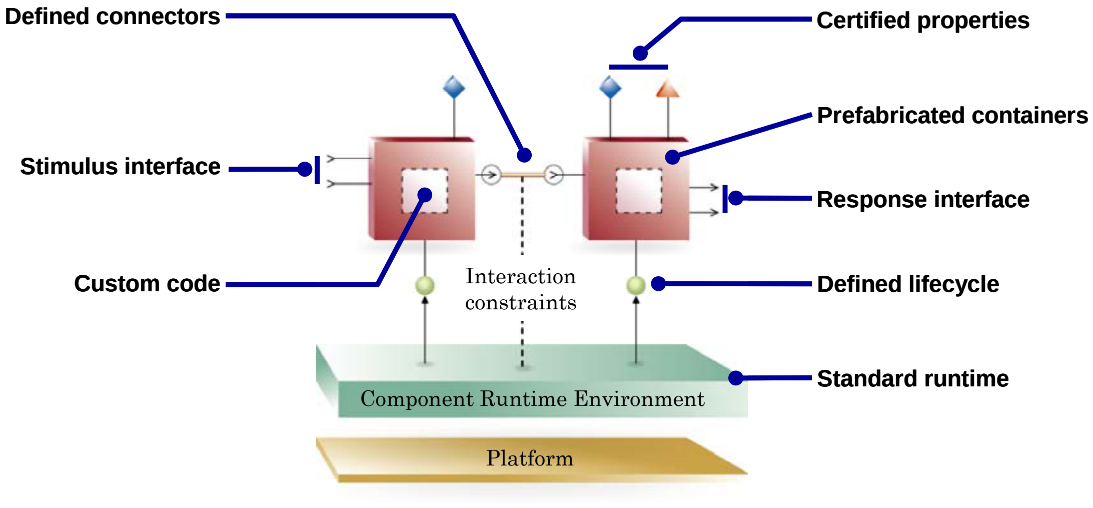


Figure 1: The Container Idiom

In this idiom, custom software code is deployed into prefabricated containers [Ward-Dutton 00]. A container restricts the visibility of custom code to its external environment and to the custom code from the environment.[2] A software component in this idiom is a container combined with custom code. Components are strictly reactive: they react only to stimuli received through the container interface and respond only through the container interface. A component runtime environment provides coordination mechanisms (or “connectors”) and implements other policies for managing resources shared by components.

The idiom does not describe a particular component technology: many implementations of the idiom are possible and simple implementations can often be realized. What matters is that container types, connector types, runtime environment, and an ability to place constraints on allowable patterns of component interaction can all be used to encode, or package, smart constraints. Moreover, the small number and uniformity of the abstractions in this idiom considerably simplify the task of automating substantial portions of the construction and prediction process—to yield predictability by construction.

2.1 Predictable by Construction

The idea of predictability by construction is simple and best understood by analogy. A Java or C# compiler checks that programs are well-formed. One check is that a program satisfies the type theory of the programming language. If this constraint is satisfied, then the compiler guarantees certain properties of program execution (technically, safety properties).

In this context, at the level of assemblies of components instead of at the programming language level, the same idea is applied. In place of type theories, we have behavior theories for nonfunctional runtime qualities such as performance. In place of specifications in a programming language, we use specifications in an architecture description language (Construction and Composition Language [CCL]) [Wallnau 03b]. In effect, CCL formalizes the container idiom and makes automated prediction and code generation possible. The result is predictability by construction. If specifications in CCL are well-formed according to the container idiom and satisfy additional reasoning-framework-specific constraints, the systems they specify will be predictable by construction. The ultimate expression of predictability by construction is to build only systems whose behaviors can be predicted.[3]

Projects such as Pervasive Component Systems (PECOS) have already exploited the affinity of component technology with predictable nonfunctional behavior [Nierstrasz 02]. However, no previous work has generalized these ideas to multiple nonfunctional attributes or emphasized the role of validation and certification to the extent reported in *Predictable Assembly of Substation Automation Systems: An Experiment Report* [Hissam 02].

[2] Different types of containers can play different roles in a global (architecture-defined) coordination scheme. An example is the sporadic server container described in this technical note.

[3] In the same way—to conclude the analogy—that the Java or C# compiler will successfully compile only those programs that are type safe (and therefore well formed).

2.2 Certifiable Quality

Analytic theories reveal which properties of the software must be known for its behavior to be predictable. A component technology imposes a standard packaging of software that includes how components are specified and what details about a component implementation must be exposed by component suppliers. Taken together, these details provide a practical basis for establishing objective quality standards for third-party software.

As an illustration, consider the prediction of the timing behavior of component assemblies using real-time queuing theory [Lehoczky 96]. Among other things, this performance theory assumes

- a scheduling discipline such as earliest deadline first (EDF)
- the identification of schedulable entities such as threads
- the first two moments of the arrival and service time distributions are known for each stream of messages

The first property is satisfied by the component runtime environment; the second, by containers. The third, however, must be satisfied by the component supplier.

Two points are worth noting from this illustration. First, an accurate measurement of the moments for arrival and service time distributions is needed; this corresponds to the *certified properties* in Figure 1. While it might be desirable, imposing requirements on the values these measurements may take is considered a separate issue. Second, the performance theory is required to give a precise definition of the measure; it may also provide strong guidance on the measurement process itself.

3 Containers Ensuring Performance in Critical Industrial Applications

In this section, we report on the application of smart constraints using the container-based idiom to a problem in industrial automation (specifically, industrial robotics) having performance-critical (specifically, timing-behavior) runtime requirements.

3.1 Industrial Robot Controller

A question faced by a manufacturer of industrial robot controllers was this: could its software controller be safely extended by third-party software having stochastic execution behavior, while also guaranteeing best service to the extension without jeopardizing controller deadline satisfaction?

The software controller can be thought of as a number of parallel, intercommunicating threads of execution within the core controller platform. That platform typically consists of a single Intel Celeron processor running VxWorks and is referred to as the *main computer*. The main computer communicates with one or more computers called the *axis computers*.

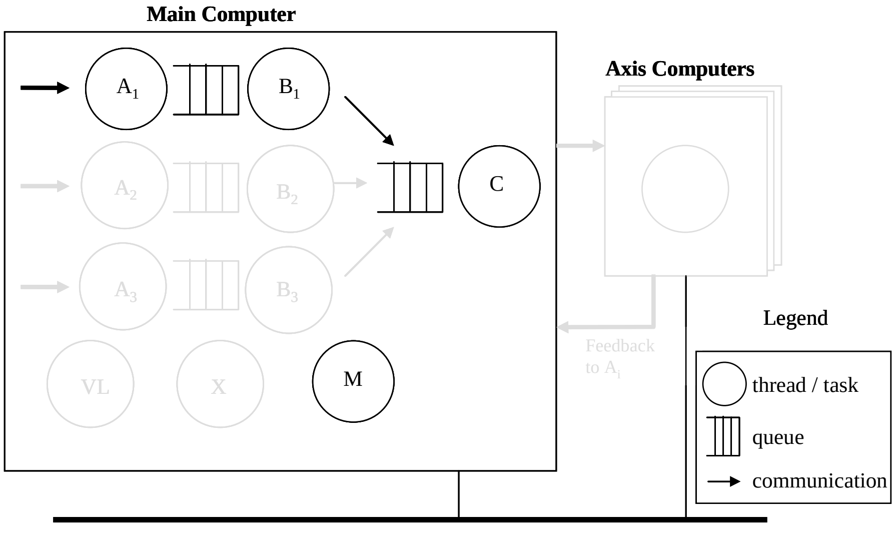


Figure 2: Tasks on the Main Computer

The problem faced by the manufacturer focuses on the interaction between tasks in the main computer. The main computer is responsible for running programs (written in a high-level robot programming language) that generate work orders.[4] The work orders are decomposed into subwork orders that ultimately result in the communication of microcoordinates to the axis computer that contains the device drivers responsible for the actual movement of the robotic arm (or arms).

The threads that execute on the main computer are a mix of periodic[5] and aperiodic[6] tasks, with some performing either synchronous or asynchronous inter-thread communication. In the context of this problem, much of the asynchronous interthread communication of interest is conducted through first in, first out (FIFO) queues. Third-party software extensions (task M in Figure 2) are envisioned to be separate threads of control.

For this problem, only a subset of the threads housed on the main computer is deemed critical (specifically, tasks A_1, B_1, C, and M). Those threads of control are shown in Figure 2 and their relevant performance characteristics are summarized in Table 1. Periodic tasks are characterized as having constant interarrival times. Aperiodic tasks have random interarrival times following an exponential distribution.

Table 1: Performance Description of Robot Problem Tasks

Task	**Priority**	**Arrivals**	**Execution Time**
A_1	Low	Exponentially distributed with mean 75 ms	Exponentially distributed with mean 9 ms
B_1	High	Constant 24 ms	Uniformly distributed 1–2 ms
C	Very High	Constant 4 ms	Uniformly distributed 0.5–1.0 ms
M	Medium	Exponentially distributed with mean 100 ms	Uniformly distributed 15–25 ms

The details of the complete problem are described further in *A Model Problem for an Open Robotics Controller* [Hissam 04a].

[4] It is not critical to know specifically what is in the program or what a work order is. It is sufficient to know that the program consists of one or more commands to a robot (much like setting a goal, such as “move here at this speed”) that are broken down into one or more subwork orders (e.g., steps to achieve that goal).

[5] A periodic task implements the response to a periodic event (one of a sequence of events having constant interarrival intervals) and thus becomes ready to execute at fixed intervals [Klein 93].

[6] An aperiodic task implements the response to an aperiodic event (one of a sequence of events *not* having constant interarrival intervals).

3.2 Container for Third-Party Software Extensions

To answer the manufacturer's question, we developed the λss reasoning framework ("λ" for latency, "ss" for sporadic server) [Hissam 04b]. As its central smart constraint, the λss framework assumes the use of sporadic server containers. The sporadic server container implements the application-level protocol for the sporadic server scheduling algorithm (SSSA) [Sprunt 89] and enforces the constraints assumed to exist by the λss reasoning framework. This is done to ensure that potential bursts of stochastic behavior are no more invasive on the periodic portions of a system than an equivalent periodic task with similar performance characteristics. The SSSA protects periodic tasks with hard deadlines from bursts of high-priority stochastic events that trigger high-priority processing by other tasks. The hallmark of the SSSA is that it creates a periodic virtual processor within which stochastic events can be processed and predictably analyzed.

Adapted from González Harbour's work, Figure 3 depicts the general behavior of a task following the SSSA [González Harbour 91]. The SSSA can be implemented in an operating system's scheduler (e.g., kernel mode) [Shi 01] or within an application (e.g., user mode) [González Harbour 91]. The container described in this report follows the latter; that is, container threads execute at the user level.

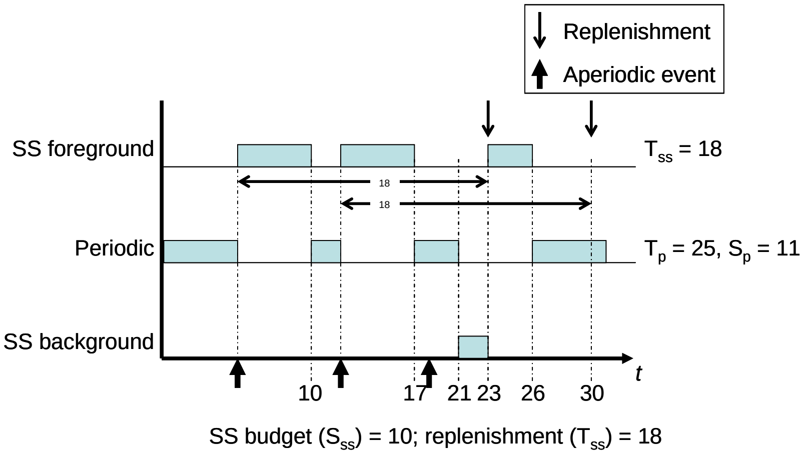


Figure 3: Example of a Sporadic-Server-Controlled Task

In this example, each aperiodic event takes five units of time to be serviced. The first two aperiodic requests arrive at *t*=5 and *t*=12 and are serviced immediately because the sporadic server starts with an execution budget of 10 units. At *t*=5, the budget of the sporadic server is decreased by five units of time. That decrease leaves a remaining execution budget of five units, enough to permit the sporadic server to execute at foreground priority. Also at *t*=5, a replenishment event is scheduled for *t*=23 (i.e., 23 = event occurring at 5 + replenishment period of 18). At *t*=12, the execution budget is again reduced by five units of time, the replenishment is scheduled for *t*=30, and the sporadic server can still execute at foreground priority. After *t*=12, the execution budget is exhausted. When the next aperiodic event arrives at *t*=18, the sporadic server is restricted to execute at background priority. The additional execution budget for five units of time is replenished at the scheduled times of *t*=23 and *t*=30, respectively, for the first two requests, thereby restoring the execution budget of the sporadic server.

To implement the SSSA at the application level (i.e., without explicit OS-level support), only two key features of the runtime environment are necessary:

1. some form of synchronous, interprocess, or interthread communication
2. the ability for one process or thread to read and change another process or thread priority

The sporadic server manager, or SSManager, is a user-level thread that operates at system high priority. The purpose of the SSManager is to manage one or more sporadic server tasks, or SSTasks, each of which processes aperiodic events. An aperiodic task can be converted into an SSTask by including two synchronous service requests to the SSManager: `arm()` and `request()`.

The high-level sequence diagrams that cover the sequence of events among the SSTask, SSManager, and the host OS are shown in Figure 4 (for `SSManager.arm()`[7] and `SSManager.request()`[8]).

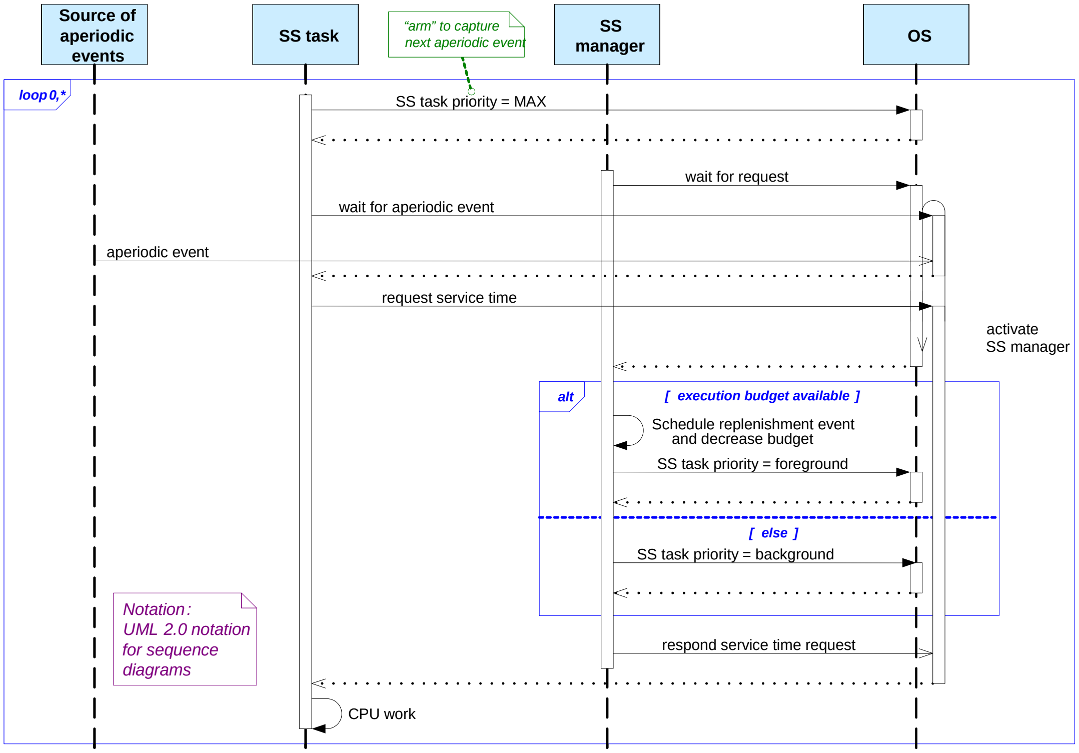
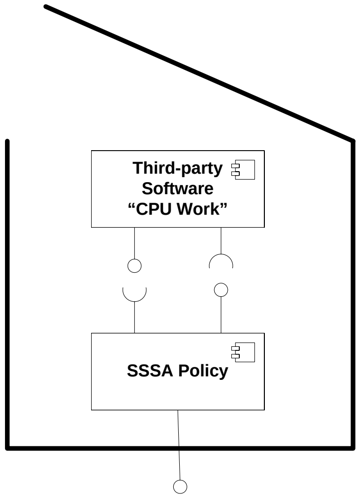

Source of aperiodic events
SS task
"arm" to capture next aperiodic event
SS manager
OS
loop 0,*
SS task priority = MAX
wait for request
wait for aperiodic event
aperiodic event
request service time
activate SS manager
alt
[execution budget available]
Schedule replenishment event and decrease budget
SS task priority = foreground
[else]
SS task priority = background
Notation: UML 2.0 notation for sequence diagrams
respond service time request
CPU work

Figure 4: UML 2.0 Sequence Diagram of Application-Level SSSA: Request and Arm

[7] As an optimization, the implementation of `arm()` bypasses the SSManager and is marshaled directly to the operating system to raise SSTask's priority to maximum.

[8] In this figure, the call to `request()` is handled via message passing using an interprocess communication mechanism routed from the sender to the receiver via the operating system.

For the industrial robot controller, the third-party custom software used to extend the controller (identified as that portion called CPU work in Figure 4) is encapsulated by a prefabricated sporadic server container. The container, provided by the industrial manufacturer, ensures that the protocol in Figure 4 is carried out correctly, restricts the visibility of the software extension to the rest of the controller's environment, and governs the extension's effects on the periodic portions of the core software controller. Using the container idiom (discussed in Section 2), the component, then, becomes the combination of the custom third-party software and the container that enforces the SSSA policy (Figure 5).



Third-party extension as a software component

Figure 5: Third-Party Extension as a Software Component in the Container Idiom

3.3 Predicting Performance for Third-Party Software Extensions

To make the analysis tractable, the problem is characterized as a single-subtask assembly [Hissam 04b] where all the periodic tasks have been collapsed into one periodic task with equivalent utilization. This approach allows the reasoning framework to consider the timing effects that the periodic portions of the problem have on the stochastic portions of the system (i.e., the third-party software extension) as a single effect. Analytically, the problem can be abstracted into a system having two tasks, as shown in Figure 6: (1) one periodic task where the periodic effects of tasks A_1, B_1, and C are collapsed into one (shown as TaskABC) and (2) one aperiodic task, TaskM.[9] TaskABC executes at a priority higher than TaskM. When TaskM is scheduled as a result of an aperiodic event, however, it will execute at a priority higher than TaskABC if an execution budget is available. If an execution budget is not available, TaskM will execute at a priority lower than TaskABC.

[9] The meanings of the variables shown in Figure 6 and the rationale for the values given are detailed in *Performance Property Theories for Predictable Assembly from Certifiable Components (PACC)* [Hissam 04b].

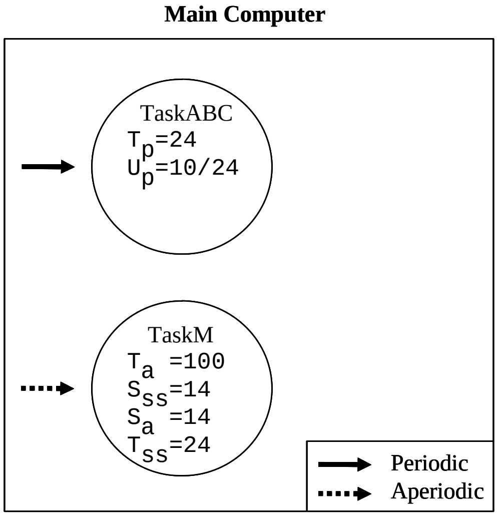


Figure 6: Analytic Representation of the Robotics Model Problem

The λss reasoning framework can generate a suite of engineering performance curves to provide insight into the expected timing behavior for the specified third-party software extension, resulting in

- best-case average latency
- worst-case average latency
- average-case latency for a periodic task given a period (shown as T_p in Figure 6) and utilization (measured in percentage of processor usage for the periodic portions of the problem; shown as U_p in Figure 6)

To serve as design guidance, the curves created by the λss reasoning framework use the performance parameters (i.e., periods, interarrival times, execution time, and budgets for aperiodic tasks) of the tasks in the system to establish bounds on the average service time of extensions. The engineering performance curves generated by the λss reasoning framework (for the analytical problem in Figure 6) are shown in Figure 7.

For this problem, the curves show that in the best case (when no periodic tasks are executing) the third-party software extension can perform its CPU work in about 15.14 ms. In the worst case (when periodic tasks are consuming nearly all available CPU resources), the extension can perform its work in about 17.79 ms. But in this specific case where the total utilization of all the periodic tasks is approximately 42% (U_p =10/24 from Figure 6) and the period of those tasks is 24 ms (T_p =24 from Figure 6), the average-case latency for TaskM is about 17.65 ms.

These curves show that it is possible to determine the best service to extensions of the software controller by providing a suite of engineering performance curves based on the specific timing parameters of all the periodic and aperiodic tasks within the controller. The theory behind the λss reasoning framework and the explanation of how it was applied to answer this question are detailed in *Performance Property Theories for Predictable Assembly from Certifiable Components (PACC)* [Hissam 04b].

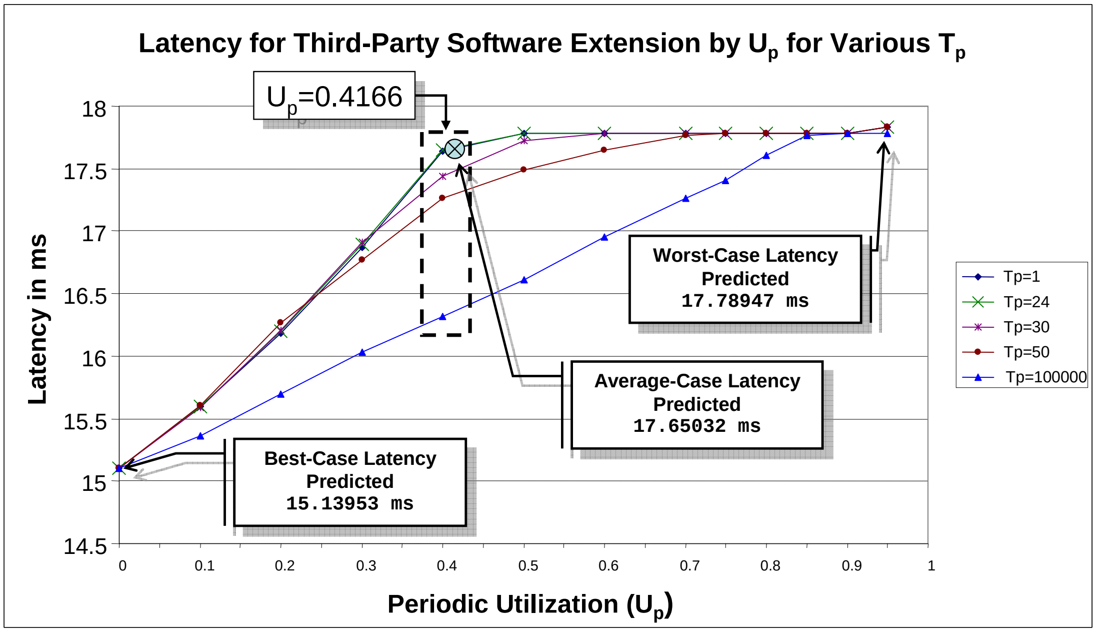


Figure 7: Engineering Performance Curves for a Third-Party Software Extension

To make those predictions, the λss reasoning framework makes assumptions about the properties of the assembly and its components. Those assumptions become the smart constraints to which the assembly (and components) must adhere. Such constraints can be satisfied by the component runtime environment, containers, specifications, or the component supplier. For the λss reasoning framework, the smart constraints and the elements that enforce those constraints are shown in Table 2.

Table 2: Smart Constraints Enforced for the λss Reasoning Framework

λss Smart Constraint	**Enforced By**
Tasks are scheduled based on priority.	Pin runtime
Tasks can be preempted by higher priority tasks.	Pin runtime
Tasks have unique priorities and do not violate priority ceiling protocol [Goodenough 88].	Assembly specification in CCL
Assemblies of tasks are confined to a single processor.	CCL
Each periodic event, whether clock- or message-based, is handled by one task (or a sequence of tasks). Each periodic task has an associated period and an execution time (or sequence of execution times).	CCL
All aperiodic events are funneled through a task managed as a sporadic server.	CCL
The sporadic server runs at the highest priority in the system and is characterized by an execution budget and a replenishment period.	Sporadic Server Container and CCL
The service time for each aperiodic event is constant and equal to the execution budget of the sporadic server.	Sporadic Server Container and third-party component supplier[10]
The aperiodic arrivals arrive according to an exponential distribution with a specified mean interarrival interval.	Assembly specification in CCL
Tasks managed as a sporadic server are allowed to use the CPU when either the sporadic server has sufficient budget or the periodic tasks are idle.	Sporadic Server Container

As illustrated in Table 2, each of these constraints can be enforced—often in more than one way. Further, such smart constraints are addressed where it best makes sense. For instance, having a preemptive, fixed-priority scheduler is best handled by the runtime environment, which is naturally designed to schedule task execution.

However, the way that a constraint can be addressed most adequately is not always obvious. For instance, it would be quite possible for a third-party component provider (e.g., the developer of an extension to the robot controller) to be required to adhere to the SSSA,

[10] For λss, it is assumed that the execution time of the supplied component is the certified execution time of the component and that the supplied component enforces that certified property.

perhaps via one or more application programming interfaces (APIs) provided by the industrial manufacturer. This approach would essentially delegate enforcement of that smart constraint to the component provider. However, it would be possible for the provider to circumvent that policy fairly easily.

To prevent such neglect (be it intentional or not), the container used to enforce the SSSA policy removes the need to delegate such enforcement to the component provider. That way, the developer of the industrial robotics application can insert the provided extension into the container needed to enforce the smart constraints required for that particular component.

4 A Container Implementation

In previous work, although it was prefabricated, the container had to be compiled with the custom code to create one component realized as a dynamic link library (DLL) [Hissam 05]. Even though this approach worked well, it had drawbacks that limited the freedom of a developer composing an assembly from components. For example, suppose a developer acquired a component that was compiled with a standard container. While designing the software assembly, the developer might conclude that it would be better to have the component adhere to the SSSA. In the container idiom, this constraint can be enforced by using a specialized container. However, in order to use the same functionality with a different container, the developer would need to obtain a new DLL from the component provider. In turn, the provider would need to recompile the same custom code for use with the new container. It is easy to see how this process can be an inconvenience in practice.

A different approach gives more freedom to developers. In the new implementation of the container idiom, the container and the custom code are packaged in separate DLLs. This allows a Pin component [Hissam 05] to be created dynamically at runtime by assembling a container and the custom code together. Also, it eliminates the need for the component provider to know a priori in which container the code is eventually going to be enclosed. As shown in Figure 8 using UML 2.0 notation [OMG 03], the custom code is encapsulated in a DLL, providing the ComponentCore[11] interface that is used by the container to invoke the custom code to respond to requests from other components or the assembly controller. In addition, the custom code DLL requires the Container interface in order to interact with its environment and other components. Mirroring these provided and required interfaces, the container DLL requires the ComponentCore interface and provides the Container interface. In addition, the container implements the Component interface through which the component as a whole interacts with its environment.

[11] In its previous versions, the ComponentCore interface was referred to as the User Code API [Hissam 05]. Since Pin is our research component technology, we have taken the liberty to change it as our ideas evolve. We plan to publish a report in the near future with updated Pin specifications.

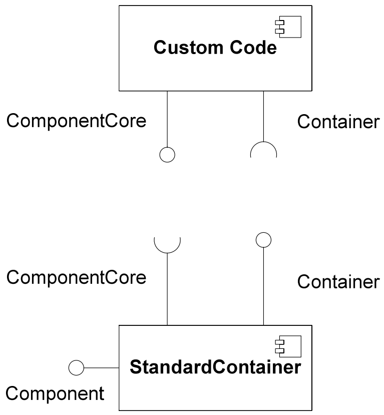


Figure 8: Parts of a Component

The dependencies between the custom code and the container have not changed with respect to the previous implementation of the container idiom [Hissam 05]. However, the implementation was changed to be able to bind their interfaces dynamically at runtime rather than at compile time. Figure 9 shows how a container and custom code are now composed to create a Pin component. Once the two parts are assembled together, they present only a Component interface, while the other interfaces are hidden.

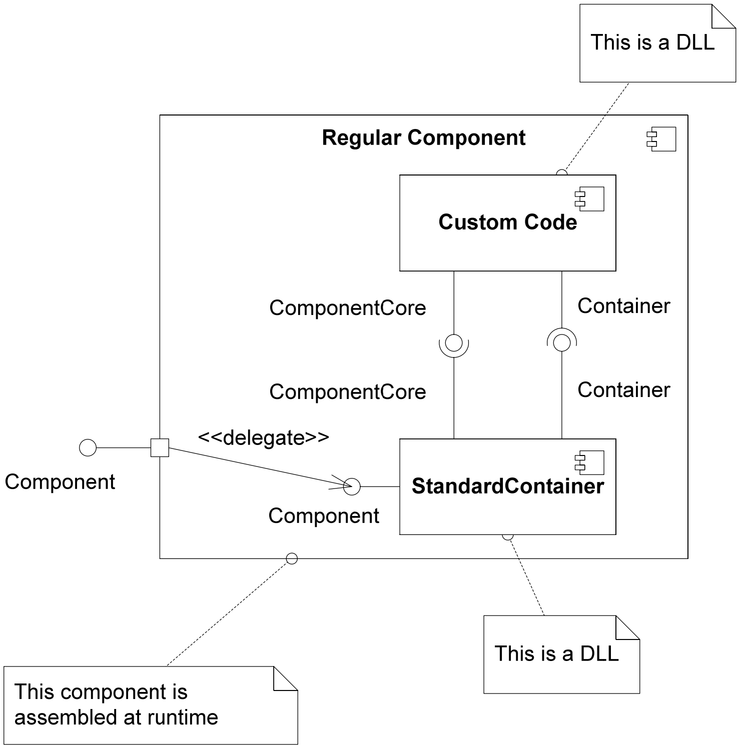


Figure 9: Component with the Container Idiom

Should the developer want to use the same custom code in a Pin component adhering to the SSSA, only a binding of the same custom code DLL with the appropriate container is needed, as depicted in Figure 10. In that way, the constraints of the sporadic server are enforced, and the component is guaranteed to satisfy the assumptions of the λss reasoning framework used to do predictions.

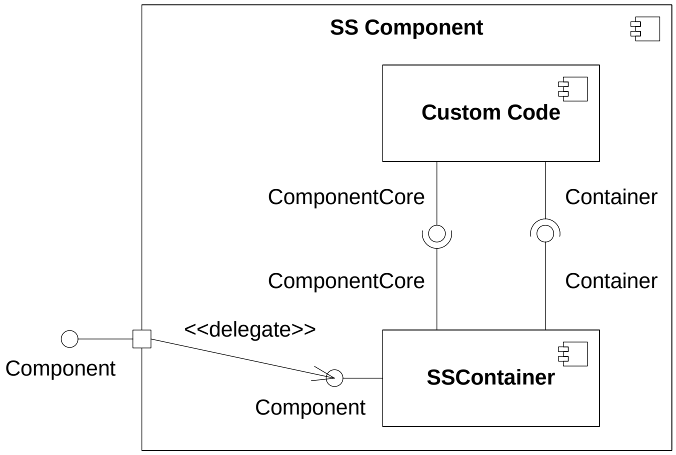


Figure 10: Using the Same Custom Code with a Different Container

The following sections describe some details of the implementation that was used to support the dynamic version of the container idiom and compare an example of its use to the original approach.

4.1 Implementation Details

In order to support the dynamic binding of containers and custom code, the assembly controller life cycle in the Pin interface was modified by inserting the steps shown in the thicker ovals in Figure 11. The first step in creating a Pin component is to load the container DLL. This is achieved with the following function in the Pin interface:

```
        TContainer* LoadContainer(char* containerName);
```

This function loads a container in memory so that it can be used to create a component dynamically, when the custom code with the following function is loaded in it:

```
        TPinComponent* LoadComponent(char* componentCoreName,
                                     TContainer* pContainer);
```

The `LoadComponent` function performs two tasks. First, it loads the custom code—also known as the component core. Second, it carries out the dynamic binding of the interfaces. The result is a Pin component ready to be instantiated. Although component instantiation and configuration are done as usual, another step that allows the container in a component instance to be configured is added. This step is optional because some containers, such as the standard container, do not require any particular configuration. However, some containers do require configuration; the SSContainer described in the next section is one example. Container configuration is done through the following function.

```
BOOL ConfigureContainer(TComponentInstance* pInstance,
                         void* pContainerData);
```

The rest of the life cycle proceeds as before [Hissam 05] except that, after components are unloaded, containers must be unloaded as well.

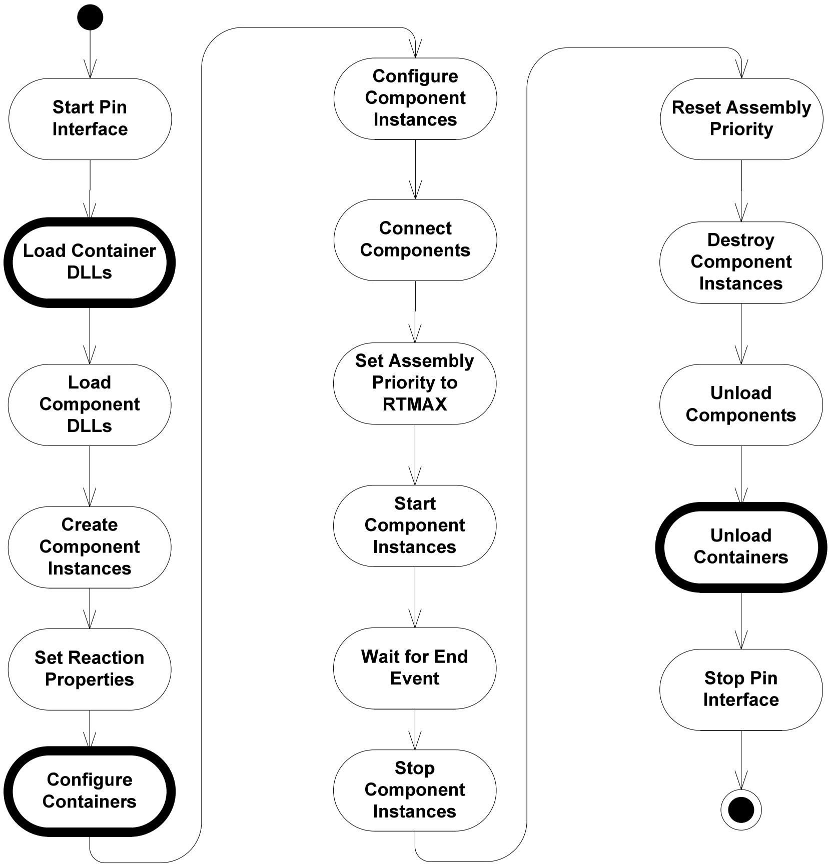


Figure 11: New Assembly Controller Life Cycle

Given that most new containers will not differ much from the standard Pin container, an inheritance mechanism for containers was implemented. The function `ExtendContainer` shown below loads a container extending a base container by overriding only those functions of the base container that are redefined in the container being loaded.

```
TContainer* ExtendContainer(char* containerName,
                             TContainer* pBaseContainer);
```

4.2 The Container at Work

In order to detail the use of containers to enforce smart constraints, this section illustrates an example based on the industrial robotics application introduced in Section 3.

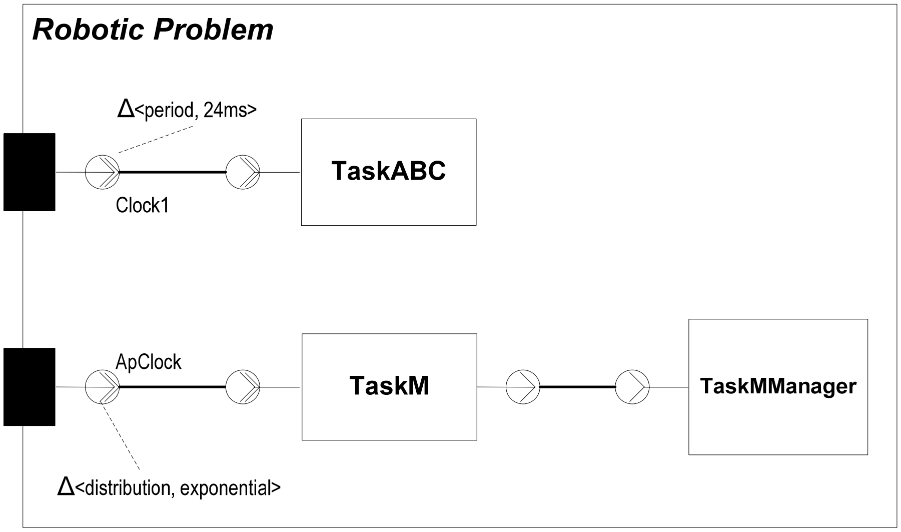


Figure 12: Pin Assembly for Analysis of Industrial Robotics Problem

We will compare two implementations of this problem that we will refer to as α and β. Implementation α was developed without using the container idiom as a vehicle for enforcing smart constraints. Implementation β was developed using the dynamic binding implementation of the container idiom, and containers were used to enforce smart constraints.

Since this assembly is only a model of a real problem and we want to analyze it with a performance reasoning framework, the components TaskABC and TaskM (shown in Figure 12) do not perform specialized functions; they only create workload (i.e., CPU work) on the processor. Execution time is given to the component instance as `properties` when the instance is created. The component TaskABC is an instance of the Synthetic component. Although TaskM must perform the same function—create workload—it must adhere to the SSSA. Therefore, TaskM has to be an instance of a different component. Implementations α and β are different because of the means used to conform TaskM to the SSSA.

In implementation α, TaskM is an instance of SSTask, which implements part of the SSSA (specifically `SSManager.request()` and `SSManager.arm()` from Figure 4). The rest (and most) of the sporadic server logic is carried out by TaskMManager, an instance of the SSManager component. This component tracks budget, controls running tasks' priorities, and performs budget replenishments. Figure 12 shows how the TaskM and TaskMManager instances are connected. Before blocking on a wait for a sink pin stimulus, SSTask sets itself to very high priority. When the stimulus arrives, it requests an execution budget from SSManager through a synchronous pin before performing its function. SSManager, in turn, sets the priority of SSTask based on the budget availability.

The following code snippet shows how the components are loaded and the instances are created and connected in implementation α. Some details have been intentionally left out to avoid cluttering the example.

```
/* load components */
pSyntheticComponent = LoadComponent("Synthetic.dll");
pSSTaskComponent = LoadComponent("SSTask.dll");
pSSManagerComponent = LoadComponent("SSManager.dll");
pClockComponent = LoadComponent("SimpleClock.dll");
pAperiodicSourceComponent = LoadComponent("DistClock105.dll");

/* create instances */
pClock1 = CreateInstance(pClockComponent, "Clock1",
                             clock1Properties,
                             sizeof(clock1Properties));
pTaskABC = CreateInstance(pSyntheticComponent, "TaskABC",
                             taskABCProperties,
                             sizeof(taskABCProperties));
pAperiodicSource = CreateInstance(pAperiodicSourceComponent,
"ApClock",
                                   apClockProperties,
                                   sizeof(apClockProperties));
pTaskM = CreateInstance(pSSTaskComponent, "TaskM",
                         taskMProperties,
                         sizeof(taskMProperties));
pTaskMManager = CreateInstance(pSSManagerComponent, "TaskMManager",
                                taskMManagerProperties,
                                sizeof(taskMManagerProperties));
/* connect pins */
SourceAddSinkPin(pClock1, 0, pTaskABC->UniqueName, 1);
SourceAddSinkPin(pAperiodicSource, 0, pTaskM->UniqueName, 1);
SourceAddSinkPin(pTaskM, 0, pTaskMManager->UniqueName, 0);
```

This implementation has two main disadvantages. First, even though the SSTask and the Synthetic components perform the same function (i.e., create workload), the same code has to be compiled in two different DLLs so that one of them complies with the SSSA. This requirement leads to maintenance issues in the best case and to other problems if the component is developed by a third party (see the discussion following Table 2). The second issue is that the approach lacks modularity. The component needs to know about the sporadic server because it has to implement part of the algorithm. Furthermore, it needs an additional source pin in order to interact with the SSManager. This, in turn, results in a more fundamental concern: the constraints of the sporadic server are not really enforced on the component; instead the developer has to trust that the component provider adhered correctly to the constraints.

In implementation β, the creation of workload is implemented in a single DLL, namely Synthetic.dll. The Synthetic component, of which TaskABC is an instance, is created at runtime by binding the custom code in Synthetic.dll with the standard container in PinContainer.dll. The complete implementation of the SSSA is encapsulated in the container SSContainer.dll; therefore, there is no need for an instance of SSManager. In order to get the same functionality provided by the custom code in Synthetic.dll and maintain compliance with the SSSA, the SSTask component is created at runtime by binding the custom code with the container in SSContainer.dll. Note that the custom code knows nothing about the SSSA. The following code shows how all this is achieved in implementation β.

```
/* load containers */
pStandardContainer = LoadContainer("pinContainer.dll”);
pSSContainer = ExtendContainer("SSContainer.dll", pGenericContainer);

 /* load components */
pSyntheticComponent = LoadComponent("Synthetic.dll",
pStandardContainer);
pSSTaskComponent = LoadComponent("Synthetic.dll", pSSContainer);
pClockComponent = LoadComponent("SimpleClock.dll",
pStandardContainer);
pAperiodicSourceComponent = LoadComponent("DistClock200.dll",
                                             pStandardContainer);

/* create instances */
pClock1 = CreateInstance(pClockComponent, "Clock1",
                            clock1Properties,
                            sizeof(clock1Properties));
pTaskABC = CreateInstance(pSyntheticComponent, "TaskABC",
                            taskABCProperties,
                            sizeof(taskABCProperties));
pAperiodicSource = CreateInstance(pAperiodicSourceComponent,
                                   "ApClock",
                                   apClockProperties,
                                   sizeof(apClockProperties));
pTaskM = CreateInstance(pSSTaskComponent, "TaskM",
                         taskMProperties,
                         sizeof(taskMProperties));
ConfigureContainer(pTaskM, &ssContainerParams);

/* connect pins */
SourceAddSinkPin(pClock1, 0, pTaskABC->UniqueName, 1);
SourceAddSinkPin(pAperiodicSource, 0, pTaskM->UniqueName, 1);
```

5 Summary

This technical note described the use of the container idiom as an effective means of packaging smart constraints to impose strict rules on visibility, coordination, and other runtime behaviors of engineered software. Smart constraints are the assumptions and invariants that must be satisfied so that reasoning frameworks can predict the behavior of their subject software systems. This approach was illustrated by predicting the average-case latency of a third-party extension introduced into an existing robot controller platform through the use of a container that enforced the SSSA. The SSSA exhibited the assumptions and invariants required by the λss reasoning framework.

Containers appear in several different component technologies. What was described here is not particular to the Pin component technology. Moreover, there are ways to enforce smart constraints in a component technology other than the container idiom. The ability to make those constraints explicit is important, because it permits us to exploit software component technology as a mechanism to encode, or package, smart constraints and allow the software system to be built predictably.

In PACC, we are investigating and documenting the use of component technologies to package smart constraints. In future work, we expect to introduce additional containers and interaction types to satisfy the invariants of more general reasoning frameworks and of those frameworks needed to predict quality attributes beyond performance.

References

URLs are valid as of the publication date of this document.

[Boehm 78] Boehm, B.; Brown, J.; Kaspar, H.; Lipow, M.; MacLeaod, G.; & Merritt, M. *Characteristics of Software Quality*. New York, NY: Elsevier North-Holland Publishing Company, Inc., 1978.

[González Harbour 91] González Harbour, M. & Sha, L. *An Application-Level Implementation of the Sporadic Server* (CMU/SEI-91-TR-026, ADA242129). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1991. http://www.sei.cmu.edu/publications/documents/91.reports /91.tr.026.html

[Goodenough 88] Goodenough, J. & Sha, L. *The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High-Priority Ada Tasks* (CMU/SEI-88-SR-004, ADA206572). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1988. http://www.sei.cmu.edu/publications/documents/88.reports /88.sr.004.html

[Hissam 02] Hissam, S.; Hudak, J.; Ivers, J.; Klein, M.; Larsson, M.; Moreno, G.; Northrop, L.; Plakosh, D.; Stafford, J.; Wallnau, K.; & Wood, W. *Predictable Assembly of Substation Automation Systems: An Experiment Report, Second Edition* (CMU/SEI-2002-TR-031, ADA418441). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu/publications/documents/02.reports /02tr031.html

[Hissam 04a] Hissam, S. & Klein, M. *A Model Problem for an Open Robotics Controller* (CMU/SEI-2004-TN-030). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents/04.reports /04tn030.html

[Hissam 04b] Hissam, S.; Klein, M.; Lehoczky, J.; Merson, P.; Moreno, G.; & Wallnau, K. *Performance Property Theories for Predictable Assembly from Certifiable Components (PACC)* (CMU/SEI-2004-TR-017, ADA431163). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents/04.reports /04tr017.html

[Hissam 05] Hissam, S.; Ivers, J.; Plakosh, D.; & Wallnau, K. *Pin Component Technology (V1.0) and Its C Interface* (CMU/SEI-2005-TN-001, ADA441815). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. http://www.sei.cmu.edu/publications/documents/05.reports /05tn001.html

[Klein 93] Klein, M.; Ralya, T.; Pollak, B.; Obenza, R.; & González Harbour, M. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston, MA: Kluwer Academic Publishers, 1993.

[Lehoczky 96] Lehoczky, J. P. "Real-Time Queuing Theory," 186–195. *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS 96)*. Washington, D.C., December 4–6, 1996. New York, NY: IEEE Computer Society, 1996 (ISBN 0-8186-7689-2).

[Meyer 03] Meyer, B. "The Grand Challenge of Trusted Components," 660–667. *Proceedings of the 25th International Conference on Software Engineering (ICSE)*. Portland, OR, May 3–10, 2003. Los Alamitos, CA: IEEE Computer Society, 2003.

[Nierstrasz 02] Nierstrasz, O.; Arevalo, G.; Ducasse, S.; Wuyts, R.; Black, A.; Muller, P.; Zeidler, C.; Genssler, T.; & van den Born, R. "A Component Model for Field Devices," 200–209. *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment (CD'2002)*. Berlin, Germany, June 20–21, 2002. Berlin, DEU: ACM, 2002.

[OMG 03] Object Management Group. *UML 2.0 Superstructure Specification: Final Adopted Specification.* http://www.omg.org/docs/ptc/03-08-02.pdf (2003)

[Shi 01] Shi, W. *Implementation and Performance of POSIX Sporadic Server Scheduling in RTLinux* (TR-010602). Tallahassee, FL: Florida State University, 2001. http://websrv.cs.fsu.edu/research/reports/TR-010602.ps

[Sprunt 89] Sprunt, B.; Sha, L.; & Lehoczky, J. *Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System* (CMU/SEI-89-TR-11, ADA211344). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1989. http://www.sei.cmu.edu/publications/documents/89.reports/89.tr.011.html

[Wallnau 03a] Wallnau, K. *Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC)* (CMU/SEI-2003-TR-009, ADA413574). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports/03tr009.html

[Wallnau 03b] Wallnau, K. & Ivers, J. *Snapshot of CCL: A Language for Predictable Assembly (*CMU/SEI-2003-TN-025, ADA418453). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports/03tn025.html

[Wallnau 04] Wallnau, K. *Software Component Certification: 10 Useful Distinctions* (CMU/SEI-2004-TN-031, ADA430991). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents/04.reports/04tn031.html

[Ward-Dutton 00] Ward-Dutton, N. “Containers: A Sign Components are Growing Up.” *Application Development Trends* (January 2000): 41–46.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY
(Leave Blank)

2. REPORT DATE
August 2005

3. REPORT TYPE AND DATES COVERED
Final

4. TITLE AND SUBTITLE
Using Containers to Enforce Smart Constraints for Performance in Industrial Systems

5. FUNDING NUMBERS
FA8721-05-C-0003

6. AUTHOR(S)
Scott A. Hissam, Gabriel A. Moreno, Kurt C. Wallnau

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-2005-TN-040

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified/Unlimited, DTIC, NTIS

12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

Today, software engineering is concerned less with individual programs than with large-scale networks of interacting programs. For large-scale networks, engineering problems emerge that go well beyond functional correctness (the purview of programming) and encompass equally crucial nonfunctional qualities such as security, performance, availability, and fault tolerance. A pivotal challenge, then, is to provide techniques to routinely construct systems that have predictable nonfunctional quality. These techniques impose constraints on the problem being solved and on the form solutions can take. This technical note shows how smart constraints can be embedded in software infrastructure, so that systems conforming to those constraints are predictable by construction.

14. SUBJECT TERMS
predictable assembly, component technology, smart constraints, reasoning frameworks, performance

15. NUMBER OF PAGES
36

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL