Software Engineering Institute

Overview of the Lambda-* Performance Reasoning Frameworks

Gabriel A. Moreno
Jeffrey Hansen

February 2009

http://www.sei.cmu.edu

Carnegie Mellon

Table of Contents

List of Figures

List of Tables

Acknowledgments

Abstract

The Predictable Assembly from Certifiable Code (PACC) Initiative at the Carnegie Mellon Software Engineering Institute is developing methods and technologies to enable the production of software with predictable behavior by making the application of analytic methods accessible to software engineering practitioners. The use of reasoning frameworks is a means to achieving this goal. A reasoning framework is a packaging of an analysis theory along with other important elements that are needed for its application, such as methods for creating analysis models and evaluating them.

Lambda-* is a suite of performance reasoning frameworks founded on the principles of Generalized Rate Monotonic Analysis (GRMA) for predicting the average and worst-case latency of periodic and stochastic tasks in real-time systems. Lambda-* can be applied to many different, uniprocessor, real-time systems having a mix of tasks with hard and soft deadlines with periodic and stochastic event interarrivals. Some examples include embedded control systems (e.g., avionic, automotive, robotic) and multimedia systems (e.g., audio mixing).

This report provides an overview of the Lambda-* performance reasoning frameworks, their current capabilities, and ongoing research. The Lambda-* reasoning frameworks have been implemented as a part of the PACC Starter Kit (PSK), a development environment that integrates a collection of technologies to enable the development of software with predictable runtime behavior.

1 Introduction

Our society increasingly depends on software embedded in all kinds of systems, including not only the ubiquitous cell phones and cars, but also medical devices, avionics, industrial robots and the power grid. This reliance on software puts pressure on the software industry to produce software that meets stringent quality requirements, such as safety conditions to prevent accidental overdoses by medical devices and performance requirements to assure the timely response to handle critical events in the power grid. Although theories capable of analyzing software to check the satisfaction of these kinds of quality requirements have existed for several years, they are still not widely used because typical practitioners are not trained in their application, and even if they are, applying them is time consuming. As a consequence, problems in meeting quality requirements are usually found—if they are found—by testing, and thus, late in the development lifecycle. In general, this implies costly rework and possible schedule and budget overruns.

The Predictable Assembly from Certifiable Code (PACC) Initiative at the Carnegie Mellon Software Engineering Institute is developing methods and technologies to enable the production of software with predictable behavior, making the application of advanced analysis methods accessible to software engineering practitioners. Making the use of advanced analysis methods accessible is a challenge itself. In the case of performance engineering, for instance, Woodside and colleagues cite the heavy effort required and the semantic gap between functional and performance concerns as some of the problems of the current practice [Woodside 2007]. The PACC approach to this problem is in packaging the expertise required to make use of analysis theories in quality attribute *reasoning frameworks* [Bass 2005]. The main elements of a reasoning framework are

- an analytic theory that is the basis of the reasoning
- analytic constraints that are based on the assumptions the theory relies on, and that define the essential invariants of systems that the reasoning framework can analyze
- a representation to capture the aspects of the system that are important for the analysis in a notation that is semantically relevant for the theory
- an interpretation that creates an analysis model from the specification of the system (architectural description, design specification, or source code), bridging the semantic gap between the software specification and the analysis model
- an evaluation procedure that evaluates the analysis model using the theory to produce the results of the prediction

A key characteristic of reasoning frameworks is that they are automated, making the details of interpretation, analytic representation, and evaluation transparent to the user. This automation supports the goal of *predictability by construction*. The behavior of a system can be predicted by a reasoning framework if it satisfies the analytic constraints of the reasoning framework. If the satisfaction of those constraints can be enforced—through statical checking of the specification of the system, through imposition of the runtime environment, or by other means—then the system

is predictable by construction.[1] In the same way that in a modern programming language certain memory safety attributes are guaranteed if a program passes the type checking, the vision of predictability by construction is to provide predictable behavior with respect to other quality attributes, such as performance or safety, if the program passes the checks for those theories.

Lambda-* is a suite of performance reasoning frameworks founded on the principles of Generalized Rate Monotonic Analysis (GRMA) for predicting the average and worst-case latency of periodic and stochastic tasks in real-time systems [Klein 1993]. Lambda-* can be applied to many different, uniprocessor, real-time systems having a mix of tasks with hard and soft deadlines with periodic and stochastic event interarrivals. Some examples include embedded control systems (e.g., avionic, automotive, robotic) and multimedia systems (e.g., audio mixing).

The Lambda-ABA and the Lambda-WBA reasoning frameworks focus on systems composed of components executing at varying priorities, and optionally communicating synchronously and asynchronously among themselves.[2] Both can handle blocking effects between components. Lambda-ABA predicts average-case latency whereas Lambda-WBA predicts worst-case latency.

The Lambda-SS reasoning framework supports predicting average-case latency of responses with stochastic (non-periodic) event interarrivals.[3] These stochastic tasks can be part of real-time systems with periodic hard deadlines because their invasiveness on the hard-real-time part of the system is bounded by the use of the sporadic server algorithm [Sprunt 1989].

The Lambda-* reasoning frameworks make the use of existing and new analysis theories (e.g., GRMA and the theory behind Lambda-SS, respectively) accessible to software engineers by providing automated generation of analysis models and their evaluation. They have been implemented as a part of the PACC Starter Kit, a development environment that integrates a collection of technologies to enable the development of software with predictable runtime behavior [Ivers 2007, 2008]. These reasoning frameworks have been validated [Moreno 2002, Hissam 2002, Larsson 2004], and have been applied to industrial systems [Hissam 2005b, Hissam 2008]. The PSK can be downloaded from http://www.sei.cmu.edu/pacc/starter-kit.html.

This report provides an overview of the Lambda-* performance reasoning frameworks, their current capabilities, and ongoing research. The report is organized as follows. Section 2 provides some background on real-time systems, and the Pin component technology and CCL, whose semantics are the foundation for the interpretation in the Lambda-* reasoning frameworks. Section 3 describes the following elements that are common to all the reasoning frameworks in Lambda-*: performance metamodel, the basic constraints, and the interpretation. Section 4 describes the specifics of Lambda-WBA, Lambda-ABA, and Lambda-SS. Section 5 describes ongoing research to extend the Lambda-* reasoning frameworks to address the predictability of systems with a mix of hard and soft real-time tasks using Real-Time Queueing Theory (RTQT) [Doytchinov 2001].

---

[1] *Predictable* does not mean that the system behaves as required. The prediction could be for example, that it is not going to meet a timing requirement.

[2] The names of the reasoning frameworks originated from the names used to represent their capabilities while they were evolving [Hissam 2002]. *Lambda* was used for latency, *ABA* means average-case with blocking and asynchrony, and *WBA* means worst-case with blocking and asynchrony.

[3] *SS* stands for sporadic server.

2 Background

This section provides some background required for the rest of the report. An introduction of the terms used in performance analysis is given in Section 2.1. Section 2.2 presents the basics of Pin, a component technology that supports predictability and is closely related to the Lambda-* reasoning frameworks. Section 2.3 provides a brief introduction to CCL, a language used to specify Pin components and assemblies of components.

It is important to note that the use of Pin and CCL is not a precondition to the use of the Lambda-* reasoning frameworks. For example, Lambda-WBA has been used to predict the timing behavior of a legacy real-time system that was not implemented using Pin [Hissam 2008], and has also been used in ArchE, an assistant tool for software architecture design [Diaz-Pace 2008]. However, using Pin and CCL provides high levels of confidence because the interpretation is based on the semantics of Pin and CCL. In addition, using Pin and CCL has the benefit of the automation provided by the PACC Starter Kit, where specification with CCL, analysis with the reasoning frameworks, and implementation with Pin are integrated together.

2.1 Performance in Real-Time Systems

Although performance is a quality attribute that is important to most systems, for some systems, the timing behavior is as important as the logical behavior. When the correctness of the system requires not only producing the right result but producing it at the right time, the system is called a real-time system. Klein and colleagues present a framework to describe and reason about real-time systems [Klein 1993]. Here we provide a brief summary introducing the terms that are used through the rest of the report.

The timing requirements in real-time systems are expressed relative to an event. An *event* is some sort of stimulus to which the system has to respond. An event can be environmental, such as the push of a button or data arriving from the network, or it can be timed, that is, generated at specific times or after a given amount of time elapses.

Events can also be classified according to their arrival pattern. In this dimension, events can be *periodic* if the time between arrivals is constant or *aperiodic* when it is not.

The computation that must be performed upon the arrival of an event is called the *response*. The amount of time it takes to complete the response to an event since the arrival of that event is called *response time* or *latency*.

Timing requirements, then, can be expressed as requirements on the response time. Furthermore, when a requirement imposes an upper bound on a response time, the upper bound is referred to as a *deadline*.

In the literature on real-time systems [Buttazzo 1999, Bernat 2001, Laplante 2004], timing requirements are usually classified as follows:

- *Hard*: when deadlines must be met at all times because failing to do so has severe consequences. For instance, reacting to a critical overcurrent condition to prevent damage on an electric motor has a hard deadline.

- *Firm*: when deadlines have to be met most times but occasionally missing a deadline does not have severe consequences. In addition, once the response misses the firm deadline, there is no value in completing it. For example, in live video streaming, dropping a video frame once in a while is not a big problem, and it is better to completely drop a frame that is late, than trying to deliver it late.

- *Soft*: when the value of responding to an event gradually decreases past the deadline, which is usually referred to as a soft deadline. For example, refreshing the display of some instrument in a panel may have a soft deadline of 30ms; however, should the refresh occasionally take longer, it will not cause a failure—only degraded performance.

In general, systems have a mix of the different kinds of timing requirements. For instance, a system may have a hard deadline to respond to a critical condition, but also have some monitoring function with less stringent timing requirements.

The focus of the performance analysis in the Lambda-* reasoning frameworks is to predict latency in systems with different kinds of requirements. There are three main contributors to the latency of a response that have to be accounted for to predict it:

1. *Execution*. The amount of time that the response takes to perform its computation without any interference from other tasks in the system.

2. *Preemption*. The amount of time that the response is not able to execute because the processor is being used by a higher priority task.

3. *Blocking*. The amount of time that the response is waiting—and consequently, not executing—for a shared resource to become available.

2.2 Pin Component Technology

PACC uses a strict form of component-based technology to achieve predictability by construction. In the Pin component technology applications—also called assemblies—are built by connecting components together without the need for "glue" code, an approach called *pure assembly* [Hissam 2005a]. Assemblies are built in a way that closely matches the component-and-connector view [Clements 2003] of the application's software architecture. Since the architecture of a software system largely determines the quality attributes of the system, having an implementation technology that allows a direct mapping of the architecture to the implementation greatly supports the predictable satisfaction of quality attribute requirements.

Pin components are purely reactive, that is, they execute only as a reaction to a stimulus. For this reason, the code that executes upon the reception of stimuli is called a *reaction*. The component receives stimuli through one or more *sink pins*, each of which has an associated reaction. The reaction can in turn interact with other components by emitting stimuli through the component's *source pins*. Components can interact both synchronously or asynchronously. The interaction

mode is determined by the pin type, and for a connection between a source pin and sink pin to be valid, both pins must have the same mode of interaction because the mode determines the interaction protocol between them. Reactions to synchronous sink pins can be either threaded or unthreaded, depending on whether they execute in their own thread of execution or in the caller's thread. Reactions to asynchronous sink pins can only be threaded. Figure 1 shows how Pin components are visually represented.

Key:
- asynchronous sink pin
- asynchronous source pin
- synchronous source pin
- N component
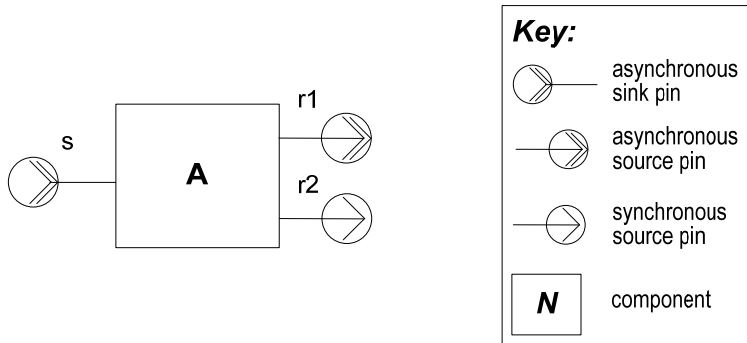
Figure 1:   Pin Component

When components interact, the stimulus that is emitted by a source pin and received by a sink pin can carry data with it. In the same way, the reply to a synchronous interaction can be accompanied by data. For this reason, pins have a signature of produced and consumed parameters. The connection between pins is valid only if their signatures are complementary (i.e., the parameters produced by one are consumed by the other and vice versa).

Pin provides *services* as interfaces to the environment. Services can have sink or source pins, so that assemblies can send stimuli to or receive stimuli from the environment respectively. Services represent, for instance, clocks, the network, the keyboard, the console, and audio devices.

Pin supports predictability in several ways. Pure assembly is important because without the need for "glue" code, all the custom code is contained in components, and these communicate through predefined connector types. Reasoning frameworks can exploit the knowledge of the semantics of the connector types so as to transform design specifications to analysis models. For example, if an asynchronous source pin is connected to many sink pins in other components, for performance analysis, it is critical to know whether the dispatching of the stimulus to each of the connected sink pins can be preempted or not because the performance model would be different for each case. If implementation decisions like this were left to component developers instead of being bound by the component technology, predictability would be much harder to achieve.

Pin uses a container idiom whereby the custom code in components is "wrapped" by prebuilt containers that mediate all the interactions of the component with other components and with its environment. Containers provide basic services that would otherwise need to be implemented in each component. For example, for the component developer, making a reaction threaded or unthreaded is a simple matter of setting a flag; the container makes the interaction within or across threads completely transparent. In addition, containers can enforce policies on the components, such as scheduling or security policies [Hissam 2005b, Moreno 2006]. Further details about Pin can be found in work by Hissam and colleagues and by Moreno [Hissam 2005a, Hissam 2005b, Moreno 2006].

2.3 Construction and Composition Language (CCL)

CCL is a language used to specify Pin components and assemblies [Wallnau 2003]. CCL specifications describe component types, with their pins and reactions, and assemblies of components consisting of component instances and the connections between them. Figure 2 shows the CCL specification of the *Hello* component, a component that upon receiving an event on its sink pin emits an event with the message "Hello world!" through its source pin. The component has an asynchronous sink pin called *go* that neither consumes nor produces parameters, and an asynchronous source pin called *message* that produces a string. The component has one reaction, *theReaction*, which is executed when a message arrives to the *go* pin. The reaction specification also indicates that it uses the *message* source pin.

```
component Hello() {
      sink asynch go();
      source asynch message(produce string s);

      threaded react theReaction (go, message) {

            // reaction code
            start -> wait {}
            wait -> say {
                  trigger ^go;
                  action ^message("Hello World!");
            }
            say -> wait {
                  trigger $message;
                  action $go();
            }
      }
}
```

Figure 2:   CCL Specification of the Hello Component

A useful characteristic of CCL is that specifications can be valid with different levels of detail. Even if the specification in Figure 2 did not include the reaction code, the specification would still be valid, and with the addition of performance-specific annotations, that level of specification would be sufficient for performance analysis. This makes it possible to predict the performance of an assembly even in the early stages of development, when only estimates of execution time are available. It also permits predicting the performance of an assembly, including components for which a complete specification is not available, for example, a third-party component. CCL allows for specifying the behavior of the reaction using statecharts and a C-like action language. This level of specification can be used to generate the implementation of the components in C language and is also required for behavior analysis [Ivers 2004].

Figure 3 shows the CCL specification of the *HelloWorld* assembly, and Figure 4 is a diagram of that assembly. This assembly prints out "Hello world!" every two seconds using an instance of the *Hello* component and two services provided by the *Rtos* environment. (Environments define the services provided by the runtime environment.) The assembly specifies in the *assume* section the services it needs from the environment to run. In this particular example, the *HelloWorld* assembly requires two services, a *Clock* and a *Console*. The assembly can only be instantiated in an en-

vironment that satisfies its assumptions. After the *assume* section in the listing in Figure 3, an instance of the *Hello* component is instantiated. Following that, the components and services are assembled together by connecting their pins with the ~> operator.

```
assembly HelloWorld() (Rtos)
{
        assume {
                Rtos:Clock clock(2000);
                Rtos:Console console();

        }

        Hello hello();

        clock:tick  ~> hello:go;
        hello:message ~> console:writeln;

        expose {}

}
```

Figure 3:   CCL Specification of the HelloWorld Assembly
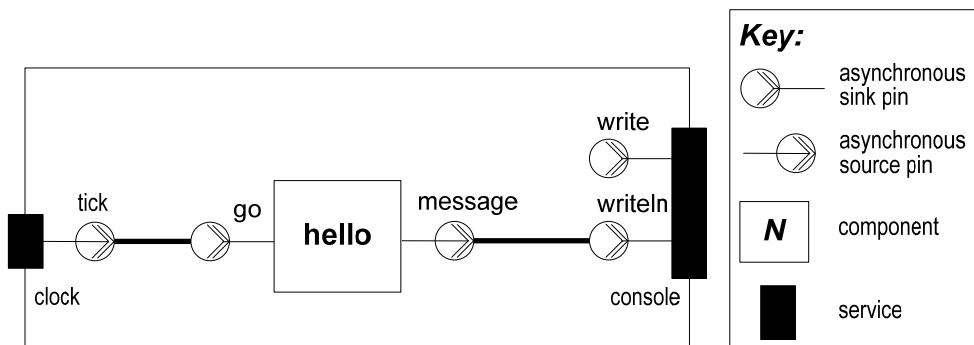
Figure 4:   HelloWorld Assembly

CCL has a mechanism to annotate different elements to provide additional information. This annotation mechanism is used to include in a CCL specification the information needed by the reasoning frameworks to carry out their analysis. For example, one of the annotations that Lambda-* requires is the execution time of the components used in an assembly. The following annotation provides that information for the computation associated with the *go* pin in the *Hello* component.

```
annotate Hello:go {"lambda*",
                const string execTime = "G(0.3, 0.5, 0.6)" }
```

The first parameter inside the block delimited by the curly braces indicates a class of annotations. In this case, it indicates that this is an annotations used by Lambda-*. The second parameter defines a typed constant, which is the specific property the annotation is providing. These properties are defined by the reasoning frameworks. In this case, *execTime* is an annotation understood by Lambda-* to indicate the execution time of the computation associated with a sink pin. The value assigned to *execTime* in this example, "G(0.3, 0.5, 0.6)" indicates a generic distribution with

minimum 0.3, mean 0.5 and maximum 0.6. For a list of the annotations supported by Lambda-*
see Appendix A.

For further details about CCL such as the specification of reaction statecharts see the technical
note by Wallnau and Ivers [Wallnau 2003].

3 Common Elements of the Lambda-* Reasoning Frameworks

This section describes the elements that are common to all the reasoning frameworks in the Lambda-* suite. Section 4 covers the specifics of each reasoning framework.

3.1 Model Representation

The performance model used in Lambda-* is based on the framework proposed by Gonzalez Harbour and colleagues [Gonzalez Harbour 1991]. In this framework, a system comprises a set of tasks that execute concurrently.[4] The work carried out by each task is represented by a sequence of subtasks that execute serially. Each subtask represents a portion of the computation that executes at a fixed priority level, does not voluntarily yield the processor, and does not access resources for which it could block. In that way, the subtask does not introduce the opportunity of a scheduling point—a point in time at which the scheduler makes a scheduling decision—in the middle of its execution. Changing the priority level, acquiring and releasing shared resources, or entering and leaving critical sections is done at the boundary between subtasks. The main attributes of a subtask are its execution time and priority level.

The metamodel for the performance models in Lambda-* is shown in Figure 5. The three classes at the top (i.e., *PerformanceModel*, *Task*, and *Subtask*) directly correspond to the aforementioned framework. That is, a performance model has a collection of tasks, and each task in turn has a collection of subtasks. The rest of the metamodel includes elements that allow modeling other situations, such as tasks that are not periodic and non-constant execution times.

The characterization of even interarrivals is done in two different ways depending on whether the task is a periodic task (*PeriodicTask*) or an aperiodic task (*AperiodicTask*). In the former, the *period* attribute in the derived class *PeriodicTask* represents the period of the task or the event that triggers the task. For the latter, the event interarrival distribution is modeled with an instance of a *Distribution*, an abstract class representing different kinds of statistical distributions.

The two most important attributes for the subtasks are the priority—a proper attribute in the class—and the execution time distribution, represented with an instance of the *Distribution* class as well. The *SSTask* represents an aperiodic task that is scheduled by a sporadic server [Sprunt 1989].

Not all the concepts in the metamodel are used by the all the reasoning frameworks in Lambda-*. For example, unbounded statistical distributions cannot be used in worst-case latency prediction. The metamodel is more general than it needs to be for any particular reasoning framework so that it can be used across the reasoning frameworks in Lambda-*.

---

4   In this report concurrent execution refers to logical concurrent execution because one assumption of the Lambda-* reasoning frameworks is the use of a single processing unit. Physical concurrency is out of the scope of this report.
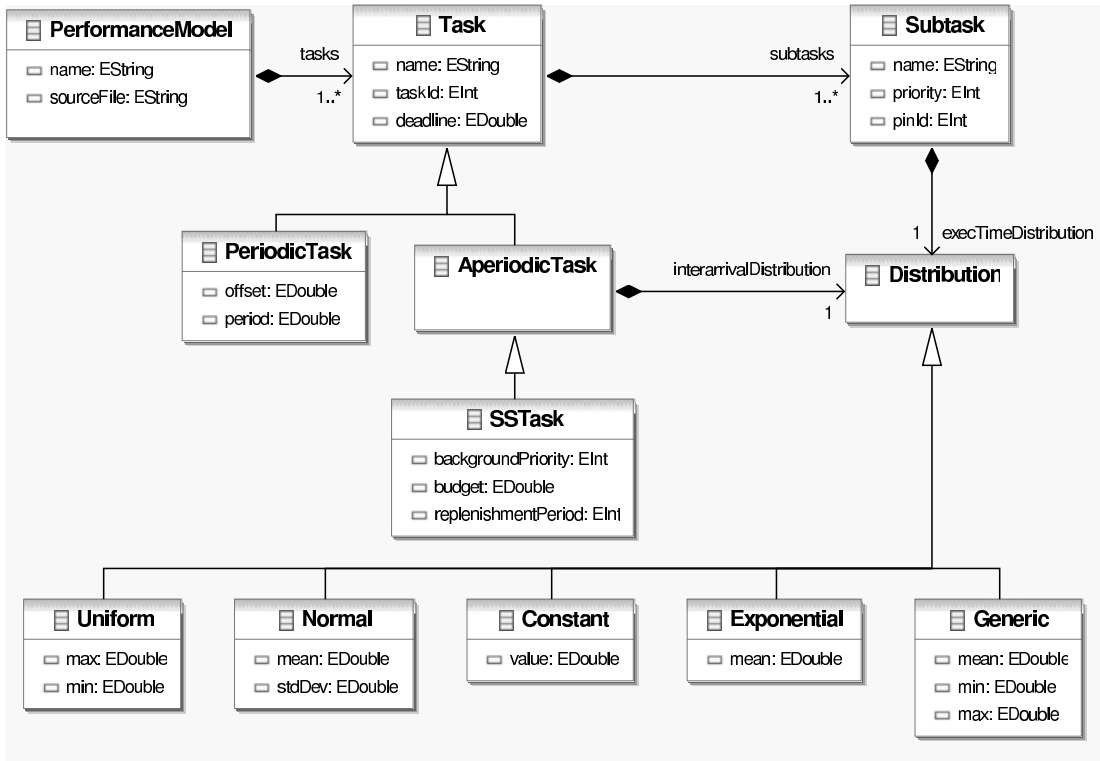
Figure 5:   Performance Metamodel (notation: UML)

3.2 Constraints

Constraints are at the heart of predictability by construction. Despite the negative connotation of the word, constraints are enablers of predictability. First, they enable the applicability of the underlying theory in the reasoning framework by ensuring that theory's assumptions are invariants of the implemented software. Second, they help make the interpretation process tractable. Unless the semantic gap between the system specification and the analysis model is sufficiently small to allow a simple direct translation (i.e., a design specification entity directly mapped to an analysis entity), the interpretation has to be able to transform known specification constructs or patterns into constructs in the analysis domain. For example, a portion of the specification stating that component *A* synchronously invokes component *B* can be interpreted as subtasks *a* and *b* in the performance model, where both belong to the same task, and *a* precedes *b*. Since the interpretation is done by applying transformations to specific patterns, the interpretation cannot handle any arbitrary specification. Constraints can delimit the specification space so that only those specifications that can be interpreted are allowed. Therefore, constraints define the space of systems predictable by a reasoning framework, ensuring that they can be both interpreted and evaluated.

The following are basic constraints of the Lambda-* performance reasoning frameworks.

1. The assembly executes in a single processing unit. This means that the assembly is not distributed and it is executed in a single core, in the case of multi-core processors.

2. The operating system uses preemptive fixed-priority scheduling.

3. Components perform their computation first and then interact with other components.

4. Each sink pin in a component requests interaction on all the source pins in its reaction.

5. There are no loops in the connection graph of components.

6. Components do not suspend themselves during their execution. That means that they do not yield the CPU by sleeping or invoking operations that could block, such as I/O.

7. Priority of mutex (i.e., synchronous non-reentrant) sink pins is assigned according to the highest locker protocol (a.k.a. priority ceiling emulation).

8. If the computations corresponding to two sink pins within the same response can be ready to execute at the same time, they must have different priorities.

These constraints exist for very specific reasons. For example, the use of preemptive fixed-priority scheduling is an underlying assumption of GRMA. The rationale behind some of the other constraints is explained in the next subsection.

3.3 Interpretation

Interpretation is the process of transforming a design specification into an analysis model expressed according to the model representation of the reasoning framework. In the case of the Lambda-* reasoning frameworks, the interpretation is from a specification in CCL to a performance model corresponding to the metamodel described in Section 3.1. The interpretation bridges the semantic gap between the design notation and the concepts of theory used to make predictions. In this particular suite of reasoning frameworks, the interpretation transforms a component-and-connector design into a model of concurrent tasks.

Although from the user's perspective the interpretation in Lambda-* is from CCL to a performance model, there is an intermediate representation used in the process. The *intermediate constructive model* (ICM) simplifies the implementation of the interpretation by making it easier to programmatically navigate the design. The ICM removes details that are specific to the design notation and makes information relevant to analysis readily accessible. For instance, to determine some characteristics of a component instance from the abstract syntax tree created from its specification in CCL, one would have to find its component type and then look for the information there, whereas in ICM, there are no types, and consequently component characteristics such as the pins a component has are directly accessible from the instance. Another example is how attributes such as pin execution times are stored. In CCL, they are given as annotations; hence retrieving their value implies searching through all the annotations of the element and even doing so at both instance and type level. In ICM, on the other hand, such properties are first-class attributes.

Another benefit of using the ICM is that the reasoning frameworks can be used with design notations other than CCL because the interpretation is isolated from a particular design language. Nevertheless, the CCL roots are obvious in ICM in that a system or application is represented as an assembly of services and components that have sink and source pins connected. The rest of this section focuses on the interpretation as the transformation from ICM to a performance model.

Figure 6 shows the ICM metamodel. The top-level class in ICM is *AssemblyInstance*, a class representing the assembly or application. An assembly has a set of services (*IcmService*) and components (*IcmComponent*), which are collectively referred to as assembly elements (*ElementInstance)*. Assembly elements have pins (*PinInstance*), each of which is either a sink pin (*SinkPinInstance*) or a source pin (*SourcePinInstance*). Sink and source pins are further specified

by their *mode* (e.g., *SinkPinMode::*asynch for an asynchronous sink, *SourcePinMode::synch* for a synchronous source). The *SSComponent* class is used to represent components that are scheduled by the sporadic server algorithm. The prediction of latency involving these components is addressed by Lambda-SS (see Section 4.3).
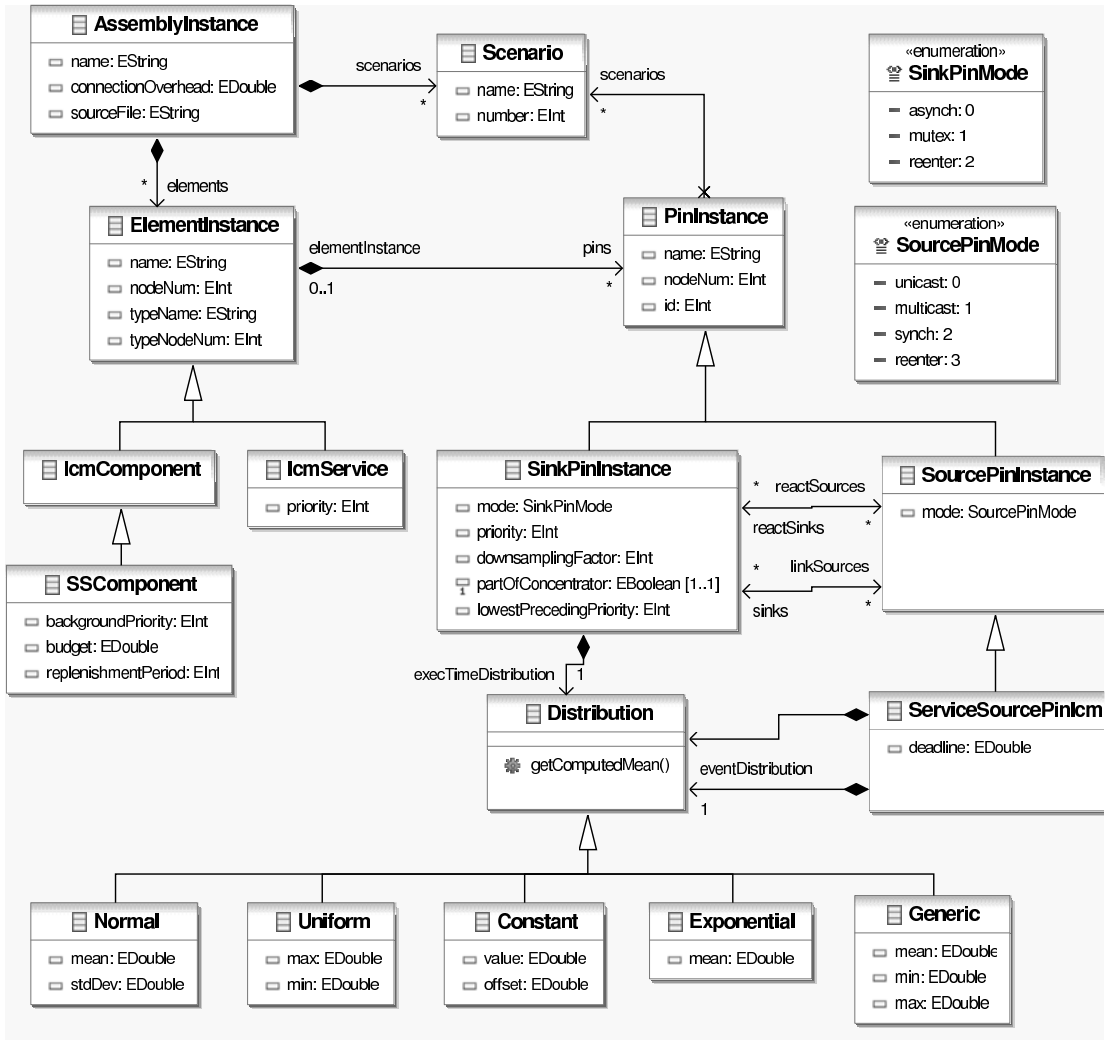
Figure 6:   ICM Metamodel (notation: UML)

CCL, and therefore ICM, is based on a reactive component model—that is components only execute as a reaction to events. The only sources of external events are the source pins on services. Since knowing the event interarrival distribution of the events the assembly has to respond to is a precondition to performance analysis, source pins in services are instances of *ServiceSourcePinIcm*, a class that requires the specification of the event distribution via its *eventDistribution* association with the *Distribution* class. The *Distribution* class is an abstract super-class to represent different kinds of statistical distributions used to describe both execution and event interarrival times. The connections between assembly elements are represented by the association *linkSources* (and the reciprocal *sinks* association) between *SourcePinInstance* and *SinkPinInstance*.

Thus far, the assembly elements and their connections have been described but nothing has been said about the inside of the components. In CCL, the inside of a component can be fully specified

by its reaction or reactions. Each sink pin is associated with a reaction that is triggered upon the activation of the sink pin. The reaction represents the computation carried out by the component. It is described as a state machine with state transitions and actions (including both computations and interaction with other components through the source pins of the components). ICM has far fewer details about the inside of the components. It has no notion of state or partial computation between interactions through source pins. Therefore, ICM makes the simplifying assumption that upon the activation of a sink pin, a computation is performed to completion, and after that the component interacts with other components via its source pins. This simplification allows the interpretation to determine the order in which different components will be scheduled at runtime, something that would not be possible if computations and component interactions were state-dependent. This is the reason for constraint number 3 described in Section 3.2.

The internals of the component are represented by two pieces of information in ICM. One is the *reactSources* association (and reciprocal *reactSinks*) that indicate those source pins that participate in the reaction corresponding to a sink pin. Again, these source pins are assumed to participate not conditionally, but always, in the reaction. The second piece of information is the execution time distribution of a sink pin (*execTimeDistribution*). This is the time the component needs to execute the computation associated with the sink pin when there is neither preemption nor blocking effects. Variations in execution time due to alternative execution paths inside the component can be accounted for by the statistical distribution used to represent the execution time.

The ICM and the performance model are different in several aspects. For example, the ICM can model a complicated network of components, while the performance model only supports concurrent sequences of activities with no explicit connections between them. The next subsections describe transformation from ICM to performance model, starting with the basic case, and then adding support for asynchronous connections and blocking.

3.3.1   Overview and Basic Case

The goal of the interpretation is to generate a performance model that supports predicting the latency of the response to an event. In an ICM, a source of events is represented as a source pin in a service (i.e., a *ServiceSourcePinIcm*). Therefore, the goal translates into predicting the latency of all the components that are connected directly or indirectly to that service source pin. Since the response to an event is modeled as a task in the performance model, it follows that for each *ServiceSourcePinIcm* in the ICM, a *Task* must be created in the performance model. The specific type of task depends on whether the event interarrival distribution of the service source pin is constant or random. In the former case the corresponding task is a *PeriodicTask* and in the latter it is an *AperiodicTask*.

The next step, and the most complex one, is transforming a response involving several components and possibly multiple threads of execution into an equivalent sequence of serially executed subtasks, resulting from determination of the order in which the components in the response will be scheduled.[5] This transformation deals with two main issues: the internal concurrency within a response and the blocking effects between responses.

---

[5]   Although a component is not exactly an entity schedulable by the operating system, it carries out a computation that executes in the context of a thread, which is a schedulable entity. Since the interpretation is a transformation from the design domain into the performance analysis domain, it is natural to have two views of the same

Before delving into these two issues, let us start with a simple assembly, shown in Figure 7, with no asynchronous connections. Since all the connections are synchronous, we can follow the call graph to determine the sequences of subtasks corresponding to the responses to the events from c*lock1* and *clock2*. The sequence for *clock1* is *<A.s, B.s, C.s>* and the sequence for *clock2* is *<D.s, E.s>*,[6] where *A.s*, for instance, refers to the subtask corresponding to sink *s* on component *A*. Each sequence can be obtained programmatically by doing a pre-order traversal of the call graph, and creating a subtask for each sink pin visited. In this case, the priorities[7] of the sink pins do not have an effect on the order of execution within one response because on synchronous interactions components block until the call is completed (i.e., the called component has completed its execution and returned the control to the caller). Therefore, even if a high-priority component calls a low-priority component, the latter will not be affected by the former. Also, the presence of threaded reactions in the components involved in a response does not introduce concurrency because components block when they call other components, so in all the cases, the last called component is the only one ready to execute.

It is worth noting that there *is* concurrency among the different tasks (or equivalently, responses) and priorities *will* have an effect on the scheduling of them. However, the interpretation only attempts to determine the order in which components will be scheduled within a response; the evaluation procedure takes care of accounting for the effect that tasks cause on one another.
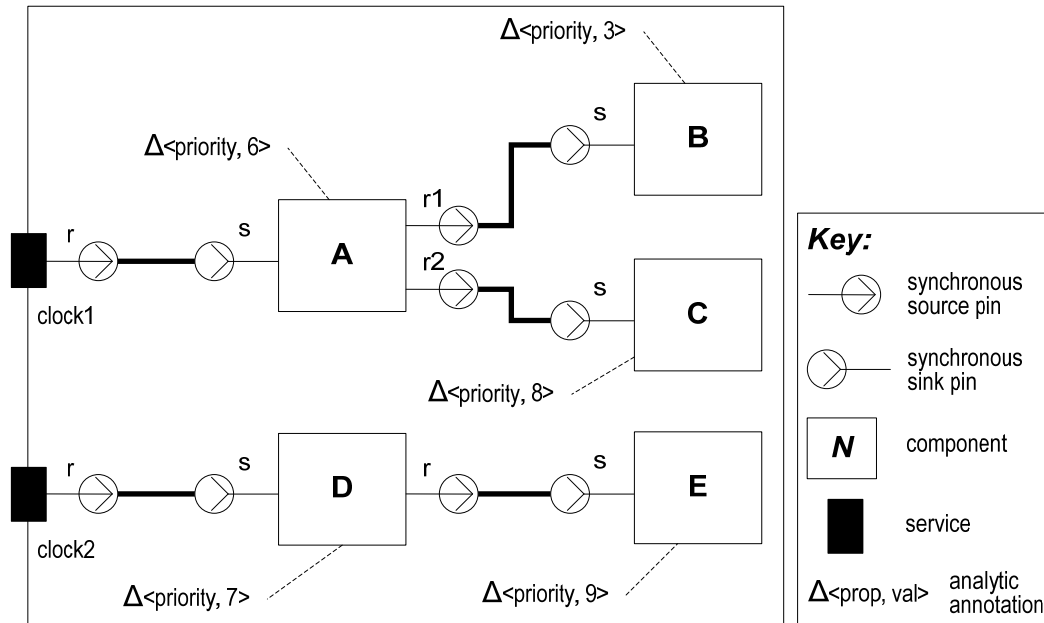

Figure 7:   Assembly with Only Synchronous Connections

behavior. For example, a component executes at priority p, and its corresponding subtask in the performance analysis domain also executes at priority p.

[6]   Source pins are used from top to bottom. For example, in component *A* in Figure 7, *A.r1* is used before *A.r2*.

[7]   A higher number indicates a higher priority.

In this example and in the rest of the discussion of the interpretation in this section, components have only one sink pin.[8] However, Pin and CCL allow having more than one sink in a component. As long as the constraints from Section 3.2 are satisfied, the interpretation algorithm works in those cases as well because it really looks at sink pins and their associated reaction as computational units, that is, it considers the sink pin and its reaction to be independent of other sinks and reactions in the component, if any. A consequence of this is that it is not possible to have two sink pins associated with the same threaded reaction. This is disallowed—indirectly—by constraint number 8 in Section 3.2. Another thing to note from this example is the omission of call unwinding in the interpretation. A component that makes a synchronous call will continue to execute after it receives control back from the callee, even if it is only to return the control to the component that called it. Due to constraint number 3 described in Section 3.2, almost all of the execution time of the component (i.e., all except calling other components and call unwinding) is consumed before it makes the calls to other components. The amount of execution time after the control returns to the component is then negligible and therefore the interpretation does not introduce a subtask to represent call unwinding in a component. Nevertheless, the interpretation takes into account the effect that the priority of call unwinding could have on the order of execution of components in the presence of asynchronous interaction.

3.3.2   Handling Asynchronous Connections

When a response includes asynchronous interactions, there is concurrency within the response because the component that initiates the asynchronous connection is ready to continue executing because

- It does not have to wait for the call to complete.
- The called component becomes ready to execute as well, possibly to invoke other components.

In this situation, the order the components will execute depends not only on the order in which they are invoked but also on the priorities at which they execute. For this reason, the preorder traversal approach to determining the equivalent sequence of subtasks no longer works.

Figure 8 shows an example of a response with asynchronous connections, and therefore concurrency. Component *A* asynchronously activates components *B* and *C*. Since *B* and *C* can execute concurrently, it seems that they cannot be serialized as a sequence of subtasks. However, because they have different priorities, and the application runs on a single processing unit (constraint number 1) they actually execute serially. The high-priority component *C* executes first followed by the low-priority component *B*, even when they are ready to execute at the same time. Therefore, it is possible to determine the sequence of subtasks that represents the way the components in the response will be scheduled at runtime, in this case <*A.s, C.s, B.s*>. On the other hand, if the components that are ready to execute at the same time in the response have the same priority, the execution order is not deterministic. Consequently, the components within a response should be assigned unique priorities. However, this is a sufficient but not necessary constraint because two components can have the same priority as long as they are never ready to execute at the same

---

[8]   For this reason, and for the sake of brevity, we can refer to the priority of the reaction associated with the one sink pin in the component as the priority of the component.

time. The interpretation algorithm flags situations where priority-level sharing is not allowed, so the necessary constraint, as expressed by constraint number 8, is that if two components within a response are ready to execute at the same time, they must have different priorities. The details of the algorithm that computes the sequence of subtasks corresponding to a response with asynchronous connection is beyond the scope of this report, but they can be found in a report by Hissam and colleagues [Hissam 2002].
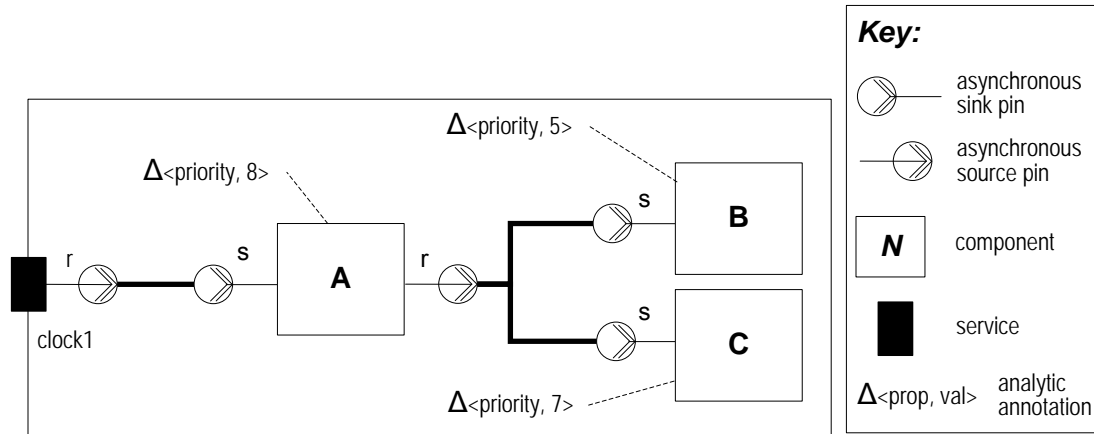
Figure 8:   Response with Asynchronous Connections and Internal Concurrency

3.3.3   Handling Blocking

The other important situation the interpretation has to address is blocking between responses, as shown in Figure 9, where two responses use a shared component *C*. Since this component is not reentrant,[9] the responses can potentially block each other. The interpretation must create one subtask in each response to represent the execution of this shared component. The resulting sequences of subtasks for *clock1* and *clock2* are <*A.s*, *C.s*> and <*B.s*, *C.s*> respectively. However, since there is no explicit synchronization between tasks in the performance metamodel, nothing would prevent the two tasks from reaching a point where both are executing the subtasks corresponding to component *C*. Since the subtasks in the two tasks would have the same priority, they could potentially execute simultaneously. That behavior in the model would not be faithful to the behavior in the assembly, where component *C* would only execute for one of the responses at a time.

The approach taken in the reasoning framework to overcome this issue was not to introduce synchronization semantics to the performance metamodel. The performance metamodel was kept simple to support different evaluation procedures. Instead, blocking is addressed using the highest locker protocol [Klein 1993], also known as priority ceiling emulation [Sha 1990, Davari 1992]. The shared component is assigned a priority higher than the priority of all the components calling it. In addition, the component must not suspend itself during its execution. The benefit of these analytic constraints is twofold. First, they make the behavior more predictable because calling components are blocked at most once and priority inversion is bounded. Second, the non-reentrant component can be modeled as a subtask in each of the responses without the need to have special

---

9   This component has a mutex synchronous sink pin, which means that this component will accept calls from other components one at a time.

synchronization elements in the performance metamodel because the highest locker protocol and the fixed-priority scheduling provide the necessary synchronization.
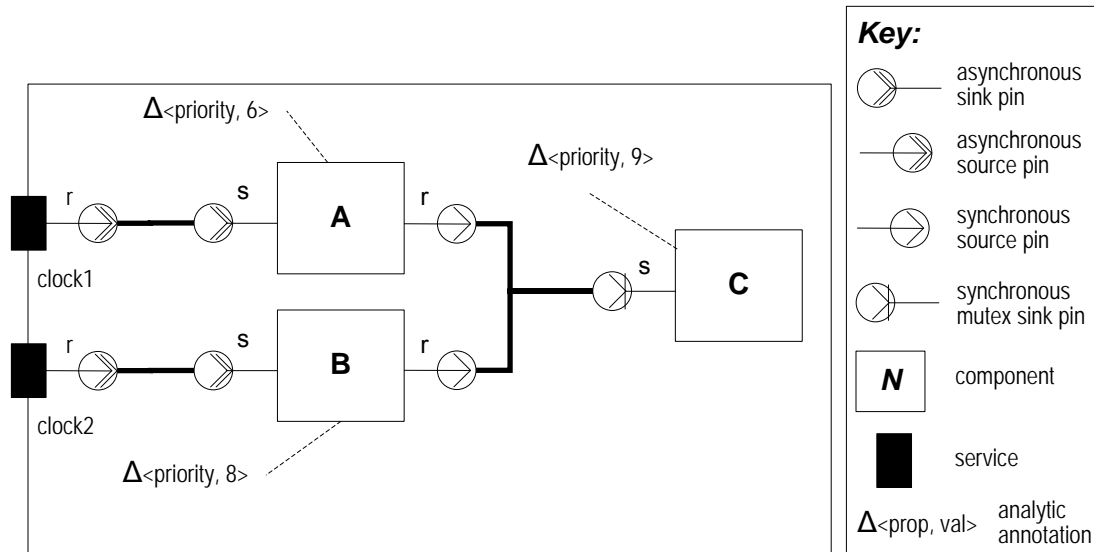
Figure 9:   Assembly with Blocking Between Responses

3.3.4   Example

We present here an example to show the performance model generated by the interpretation of a design that uses both synchronous and asynchronous interactions, and has two responses that can block each other when accessing a shared repository. This example, a simple robot controller, is based on an example included in the PACC Starter Kit and has been described elsewhere [Moreno 2008].

The design of the robot controller is shown in Figure 10. The activities in the controller are periodic, thus, they are driven by clocks. The period of the clocks is included in the name for simplicity (e.g., *clock130* is a clock with a period is 130ms). The *trajectoryPlanner* receives high-level work orders for the robot and translates them into sub-work orders, which are then put in the *repository*. For each work order, three sub-work orders are created. To do this translation, the trajectory planner needs to get information about the position of the robot, which is provided by the *positionMonitor* component. The *movementPlanner* component takes sub-work orders from the repository and translates them into low-level movement commands for the controllers that control the axes of the robot (*controllerX* and *controllerY*). It is critical that the movement planner never finds the repository empty because if it does, the robot operation has to be aborted. Therefore, the trajectory planner has a hard deadline of 450ms. At the same time, the robot must receive movement commands every 150ms; consequently the movement planner has that deadline. The remaining components in the assembly are a *sensor* that senses the position of the robot and a *monitor* component that does monitoring of the system at low priority. The CCL code for the components and their controller assembly are included in Appendix B.

Table 1 contains the additional information needed to generate the performance model, that is, priorities and execution times. The interpretation takes the design as input and generates the performance model shown in Figure 11 as a screenshot of the PACC Starter Kit. In the performance

model, each task is described as a sequence of activities without any reference to synchronous or asynchronous interactions or blocking.
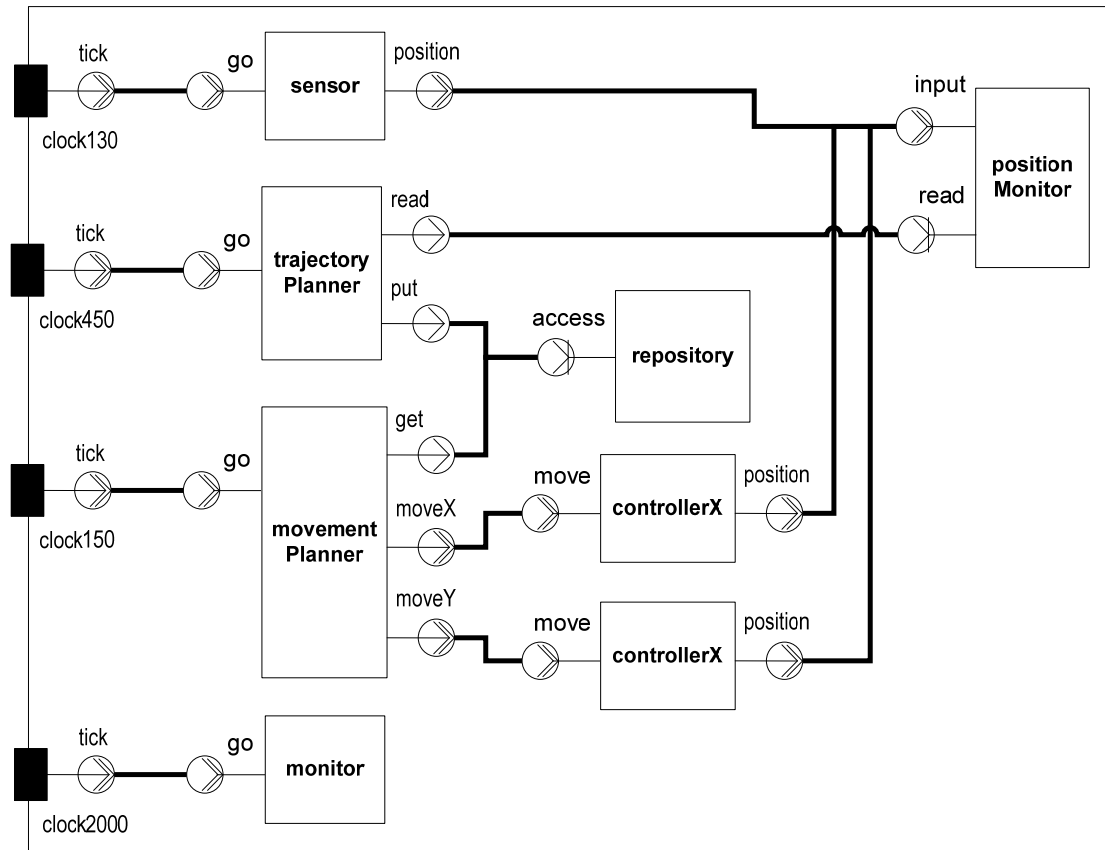
Figure 10:   Robot Controller Example

Table 1:   Priorities and Execution Times in Robot Controller

Component	Pin	Priority	Execution Time		
			Minimum	Average	Maximum
controllerX	move	20	12.8	13.0	13.5
controllerY	move	20	12.8	13.0	13.5
monitor	go	2	0.25	0.3	0.5
movementPlanner	go	16	18.8	20.0	21.00
positionMonitor	input	12	9.8	10.0	10.8
positionMonitor	read	14	3.0	3.1	3.2
sensor	go	10	5.0	5.1	5.6
trajectoryPlanner	go	4	88.5	89.5	90.5
repository	access	18	19.8	19.9	20.8
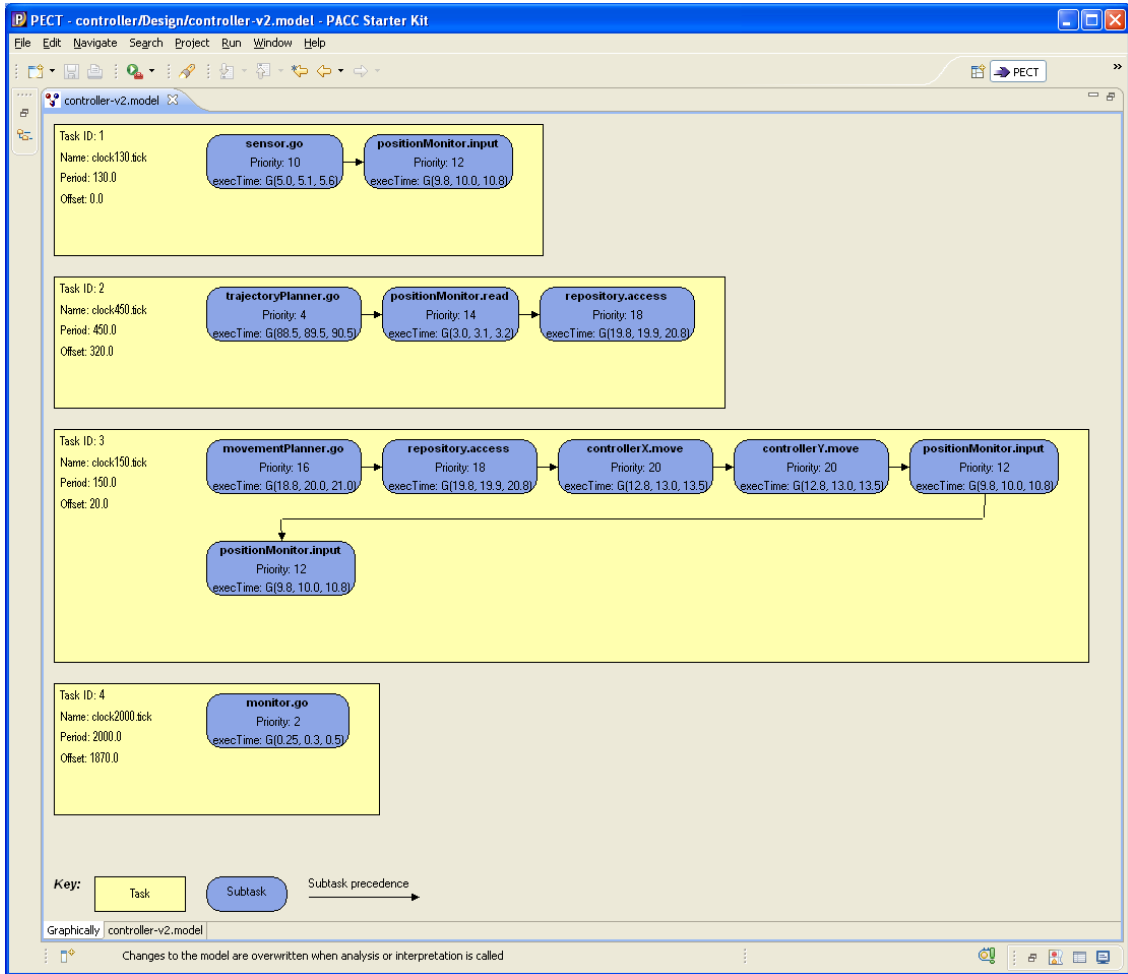
Figure 11:   Performance Model for the Robot Controller

4 Specifics of the Reasoning Frameworks in Lambda-*

The following subsections describe in detail the different reasoning frameworks in Lambda-*. Each one is described in terms of the elements of a reasoning framework, omitting those elements that have already been covered in Section 3.

4.1 Lambda-WBA: Worst-Case, Blocking, and Asynchrony

4.1.1 Measure of Interest

Lambda-WBA predicts the worst-case latency for the response to an event. The computed value is an upper-bound for the latency because the worst-case component execution times, blocking, and preemption effects are assumed to occur simultaneously. Although this may seem overly pessimistic, there are some real-time systems for which it must be assured that the responses to certain events are handled on time under all possible conditions. In these situations, Lambda-WBA can be used to predict whether the response to an event will complete before its deadline.

4.1.2 Theory

The underlying theory of Lambda-WBA is Generalized Rate Monotonic Analysis (GRMA), more specifically, a technique for analyzing the schedulability of a set of tasks with varying priorities [Gonzalez Harbour 1991]. According to this theory, each task or response is composed of a sequence of subtasks that have an associated execution time and priority level. This makes it possible to analyze situations in which the response to an event is composed of several computations executing at different priorities, which is the kind of response found in a component-based system, where each component carries out a portion of the response and can execute at its own priority level if it has its own thread of execution. In this case, the assignment of priorities can be based on deadlines or the semantic importance of the component [Gonzalez Harbour 1991]. In addition, the theory can also account for the effect of the synchronization between responses when using a priority-based synchronization protocol.

The most complex aspect of this theory involves computing the preemption effect. In regular rate monotonic analysis, each task executes at a fixed priority, so the set of tasks that can preempt the task being analyzed is constant, and they preempt every time. With the varying priorities method, priorities can vary throughout the execution of both the task being analyzed and the other tasks in the system. Therefore, the set of tasks that can preempt the task being analyzed is not constant. The algorithm for computing the worst-case latency for tasks with varying priorities classifies the other tasks in the system based on their ability to preempt each of the subtasks in the task being analyzed. First, the task being analyzed is transformed to *canonical form*, a special form of the task wherein the priority of consecutive subtasks does not decrease and that for worst-case analysis is equivalent to the original task. If P is the priority of the subtask being analyzed, the rest of the tasks are classified in the following sets:

- H: set of tasks whose lowest priority is higher than or equal to P. These tasks preempt every time (when they execute at a priority equal to P, they are assumed to preempt, the worst effect).

- HL: set of tasks that start at a priority higher than or equal to P and then drop below P. These tasks preempt only once because when they arrive they are higher priority, but once they drop to low priority they cannot complete until the task being analyzed completes.

- LH: set of tasks that start at a priority lower than P and eventually rise over P. Only one of the tasks in this set can preempt since a task from this set can only preempt if it is already executing its high-priority segment when the subtask being analyzed starts; only one of them could be executing its high-priority segment at that time.

- L: set of tasks whose priority is always lower than P. These never preempt.

The algorithm then uses these sets in the process of computing the worst-case response time of the subtask being analyzed [Gonzalez Harbour 1991, Klein 1993].

4.1.3   Constraints

In addition to the basic constraints of Lambda-*, Lambda-WBA has the following constraints.

- Only lower bounded interarrival time distributions are allowed.
- Only upper bounded execution time distributions are allowed.

Only these bounded distributions are supported because for worst-case analysis, the worst interarrival and execution times are used. If they were described by unbounded distributions, then the analysis would assume events arrive with infinite frequency and components have infinite execution time, which of course results in the impossibility to schedule the tasks.

4.1.4   Evaluation Procedure

The evaluation procedure in Lambda-WBA is the GRMA method for determining the schedulability of tasks with varying priorities [Gonzalez Harbour 1991, Klein 1993]. More specifically, the Lambda-WBA reasoning framework uses MAST [Gonzalez Harbour 2001], a worst-case analysis tool that implements that method. Since MAST has its own input language for performance models, the performance models generated by the interpretation in Lambda-WBA are translated to an input that allows MAST to evaluate the model. Nevertheless, the translation is straightforward because both the Lambda-* performance metamodel and the language of MAST are in the same domain and have very similar concepts.

The worst-case latency is computed by constructing the worst possible alignment of preemption and blocking effects for each task. It is worth noting that the resulting worst case is an upper bound that may not actually be observed in the execution of the system if it is unlikely that all those worst effects will happen simultaneously. Figure 12 shows the output of MAST for the example introduced in Section 3.3.4. The worst-case latency for each task is shown with a green or red background depending on whether the task is going to meet its deadline or not, respectively. In this case, the responses to *clock130* and *clock450* do not meet their deadlines.

Figure 12:   Results of the Lambda-WBA Evaluation Procedure with MAST

4.2 Lambda-ABA: Average-Case, Blocking, and Asynchrony

4.2.1   Measure of Interest

During the execution of a system each job of a response (i.e., each instance of a response) can be affected differently by other tasks and thus exhibit different latencies as shown in Figure 13. Lambda-ABA predicts the average latency for the response to an event by taking into account how different jobs are affected by other tasks. Instead of creating an alignment of tasks that causes the worst case for a response, as in Lambda-WBA, Lambda-ABA uses the alignment that naturally occurs from the arrival patterns and execution times of the different tasks.

4.2.2   Theory

The evaluation procedure of Lambda-ABA is based on discrete-event simulation, and consequently there is no specific theory directly used to compute the latency prediction in the reasoning framework. Nevertheless, many of the concepts in Lambda-WBA that come from the GRMA technique for scheduling tasks with varying priority [Gonzalez Harbour 1991] are used to make the simulation very efficient. For example, the highest locker protocol is used to do priority-based task synchronization. In this way, the simulation does not need to handle synchronization specifically because it is handled by virtue of its simulation of fixed-priority preemptive scheduling.

When all the sources of events in the assembly whose performance is being predicted are periodic, Lambda-ABA makes an optimization to drastically reduce the length of the simulation. When all the tasks are periodic, it is possible to find a hyper-period, defined as the least common multiple (LCM) of the periods of all the tasks as shown in Figure 13. Hyper-period analysis can be used only if the execution times of the components are constant or have a negligible variance; in other cases, looking at a single hyper-period would not allow for the varying execution times of a component to be sampled.
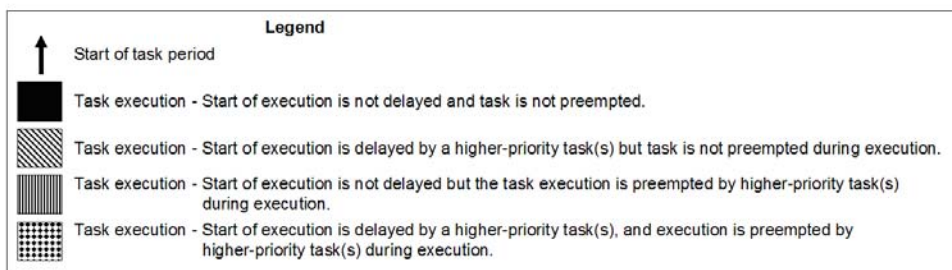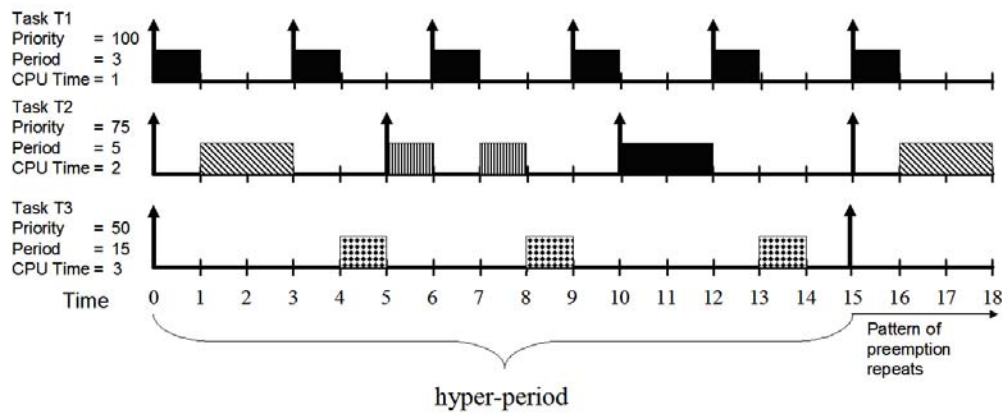
Figure 13:   Example of Hyper-period with Three Periodic Tasks

4.2.3   Constraints

In Lambda-ABA, the interpretation does not impose any specific constraints other than the basic constraints of Lambda-* (Section 3.2). However, the different evaluation procedures supported by Lambda-ABA have some constraints regarding some of the elements in the performance model that they do not support, such as a specific statistical distribution. These constraints are only due to tool limitations rather than to concrete limitations of the theory.

4.2.4   Evaluation Procedure

Lambda-ABA supports three different evaluation procedures, all based on discrete-event simulation. Simulation-based evaluation procedures make latency predictions by simulating the execution of the system, generating random event interarrival and execution times following the distributions specified in the model. While running the simulation, they keep track of best, average, and worst latency. Lambda-ABA simulates the execution of the performance model instead of simulating the execution of the system. The advantage of doing this is that the simulator does not need to handle blocking (other than the resulting from fixed-priority scheduling), nor does it need to maintain a call stack. The latter is due to the fact that the interpretation has transformed all the calls—both synchronous and asynchronous—into plain sequences of subtasks. That is, the scheduling within a task has already been done by the interpretation, leaving less work for the simulation to do. This approach results in faster simulations.

The following subsections describe the three simulation-based evaluation procedures of Lambda-ABA.

4.2.4.1   Evaluation with SIM-MAST

SIM-MAST is a discrete event simulator specially designed to simulate performance models [Lopez Martinez 2004]. It is part of the MAST suite, so it accepts as input performances models expressed in the same language that MAST uses.

The following constraints are specific to the use of SIM-MAST as an evaluation procedure for Lambda-ABA.

- Only constant, exponential, and uniform interarrival time distributions are allowed.
- Only constant, uniform, and generic[10] execution time distributions are allowed.
- Sporadic servers are not supported.

Figure 14 shows the results produced by the evaluation with SIM-MAST for the example from Section 3.3.4. The results are presented in the same way as the results of the evaluation with MAST in Lambda-WBA, with the addition of the average response time. In this case, the worst response is the worst observed in the simulation, which is not as bad as the worst case predicted by Lambda-WBA. The difference may be due to several reasons. For example, missing a deadline may require that several components exhibit their worst-case execution time simultaneously, and the simulation may have not generated such a case.

Transaction	Event	Referenced Event	Best Response	Avg Response	Worst Response	Hard Deadline
clock130.tick	O_1_2_done	clock130.tick	14.80	51.94	125.52	130.00
clock450.tick	O_2_3_done	clock450.tick	310.48	348.18	436.04	450.00
clock150.tick	O_3_6_done	clock150.tick	84.00	89.41	109.41	150.00
clock2000.tick	O_4_1_done	clock2000.tick	12.69	219.59	433.24	2000.0

Figure 14:   Results of the Lambda-ABA Evaluation Procedure with SIM-MAST

4.2.4.2   Evaluation with Qsim

Qsim is a fast discrete event simulator specially designed to simulate performance models and queueing networks. As input, it takes a network of queues with potentially different queueing policies at each queue. Jobs are introduced through "flow" declarations (representing tasks) that

---

10   Generic distributions are described by their minimum, average, and maximum values.

specify the queues through which a stream of jobs pass, and the characteristics of those jobs. In addition to specifying the interarrival and execution time characteristics, flows may also be bound to a sporadic server. Although *Qsim* supports several queue disciplines such as earliest deadline first, Lambda-ABA uses *Qsim* with a single static-priority queue to simulate the fixed-priority scheduling.

At the end of the simulation, Q*sim* reports the minimum, mean, and maximum latency for each flow, corresponding to the latency predictions for each task in the performance model generated by Lambda-ABA.

4.2.4.3   Evaluation with Extend

Extend is a commercial general-purpose simulation tool supporting continuous and discrete-event models [Krahl 2001]. Extend simulation models can be created by visually adding pre-built components, setting their properties, and connecting them together. In addition to the components in the standard libraries, new components can be created with its ModL language. To simplify the creation of simulation models, a library of components was created especially for Lambda-ABA, including components such as *Task*, *Subtask*, and *CPU*. The evaluation procedure then takes the performance model created by the interpretation and translates it to an Extend model by programmatically instantiating components from the library and connecting them together.

When using the Extend-based evaluation procedure in Lambda-ABA, the following constraints apply.

- Only constant execution times are used. When other distributions are used for execution time, the average is used as a constant execution time in the generated simulation model.
- Only constant, uniform, normal and exponential interarrival time distributions are supported.
- Explicit deadline annotations are not supported.

4.3 Lambda-SS

4.3.1   Measure of Interest

Lambda-SS predicts the average latency for the response to an event when the response is carried out by a component scheduled through use of the sporadic server (SS) algorithm [Sprunt 1989].

4.3.2   Theory

The sporadic server scheduling algorithm provides a solution for scheduling aperiodic tasks in a system (i.e., tasks that respond to stochastic events). The need to respond to stochastic events often presents a dilemma. Due to the nature of the events, they could arrive in bursts, creating a high demand for processor time. If the response to the event is assigned a high priority, then it could prevent periodic tasks with hard deadlines from meeting those deadlines. To avoid that, the aperiodic tasks could be assigned a low priority that would prevent them from having an unbounded negative effect on the periodic part of the system. However, this would also relegate the aperiodic tasks to executing only in the background, that is, when no periodic tasks are executing or ready to execute.

The sporadic server algorithm provides a solution to this problem by reserving a budget of high-priority execution time to be used by aperiodic tasks. When an event to be handled by an aperiodic task arrives, the sporadic server schedules the task to be run at high priority if there is execution budget left; otherwise it runs at background priority. In the former case, the budget is scheduled to be replenished one *replenishment period* later by an amount equal to the execution time consumed by the task. When the budget is replenished, if there are aperiodic tasks ready or running at background priority, they are promoted to high priority so that they can benefit from the reserved high-priority execution time that has just become available.

The sporadic server algorithm provides a good quality of service to the aperiodic tasks and at the same time bounds their invasiveness on hard real-time periodic tasks in the system. In fact, when analyzing the hard real-time periodic part of the system, the aperiodic task scheduled by the sporadic server can be considered as a periodic task with execution time equal to the sporadic server budget and period equal to its replenishment period. That is, it retains the predictable timing behavior of the periodic part of the system, which can still be analyzed with Lambda-WBA. Lambda-SS focuses on the predictability of the aperiodic task in the sporadic server.

The theoretical underpinnings of Lambda-SS are reported by Hissam and colleagues [Hissam 2004]. Here, we present a short summary of its key concepts. Lambda-SS builds on queueing theory to predict the latency of the response to a stochastic event. Basically, the expected or average latency $E[W]$ can be computed as the sum of the mean queueing time $E[Q]$ and the mean service time $E[S_a]$ as shown in Equation (1).

$$E[W] = E[Q] + E[S_a] \qquad (1)$$

Assuming exponentially distributed interarrival times, the mean wait time can be determined using the Pollacek-Khinchin formula [Kleinrock 1975] as shown in Equation (2):

$$E[Q] = \left(\frac{\rho}{1-\rho}\right)\left(\frac{E[S_a^2]}{2E[S_a]}\right) \qquad (2)$$

where $\rho = E[S_a]/E[T]$, in which T is the mean interarrival time of the aperiodic events. It is obvious then, that to compute the mean wait time, the mean service time is needed. However, the service time of the aperiodic task in the sporadic server depends on the amount of high priority execution budget available during its execution. An important result presented by Hissam and colleagues [Hissam 2004] allows us to determine the mean service time from the point of view of the queue, that is, the one needed for Equation (2), in a special case called *continuous background*. Once the sporadic server budget is exhausted, the aperiodic task can execute when the periodic tasks are not executing, that is, at low priority in background.

For example, if there is one periodic task with execution time 8ms and period 10ms, background execution time will be available for 2ms every 10ms. If the period of the periodic is reduced while keeping the same utilization, for instance execution time 0.2ms and period 1ms, background is available in smaller chunks but more often. If this is taken to the extreme of having an infinitesimal period for the periodic task, background becomes available for infinitesimal periods of time infinitely often, hence the name continuous background. From the point of view of the aperiodic

task, it looks as if it were executing in a slower processor, and its "degraded" service time can be computed as in Equation 3:

$$\hat{S}_a = \frac{S_a}{1 - U_p} \tag{3}$$

where U_p is the utilization of the periodic tasks.

What is equally important is that from the point of view of the events waiting to be serviced in the queue, the apparent service time of the aperiodic task is always the one given by Equation 3, regardless of whether the task is executed completely in the sporadic server at high priority, completely in background, or somewhere in between—a case called *hybrid*. This is because even if it executes at high priority, the task waiting in the queue still has to wait for the backlog of periodic work to be worked off before it can be executed. See the report by Hissam and colleagues for the proof [Hissam 2004].

Having taken care of the first term in Equation (1), the rest of the theory is concerned with computing the mean service time for the aperiodic to use in the second term of Equation (1). This requires computing the distribution of sporadic server, background, and hybrid arrivals, and also the distribution of high-priority execution in the latter. This is done drawing from results of queueing and renewal theory.

In addition to providing a way to determine the average latency of the aperiodic task in the continuous background case, Lambda-SS provides a way to compute the average latency at the other end of the spectrum, when the period of the periodic task is significantly large [Hissam 2004].

4.3.3 Model Representation

The performance metamodel shown in Figure 5 has a class named *SSTask* specifically used to model aperiodic tasks that are scheduled by a sporadic server. The attributes of this class correspond to the parameters of the sporadic server, namely, execution budget, replenishment period, and background priority.

4.3.4 Constraints

In addition to the basic constraints of the Lambda-* reasoning frameworks, Lambda-SS has the following constraints.

- There is exactly one sporadic server task.
- The rest of the tasks are periodic.
- The interarrival distribution of the sporadic server task is exponential.
- The sporadic server task must have constant execution time.
- The sporadic server budget must be equal to the sporadic server task execution time.
- The background priority of the sporadic server is lower than the priority of any periodic subtask.

4.3.5 Interpretation

The interpretation is the same as for the rest of the reasoning frameworks in Lambda-* with the addition of a post-processing step before the evaluation that computes the total utilization of the periodic tasks, a parameter needed for the evaluation.

4.3.6 Evaluation Procedure

The evaluation procedure for Lambda-SS has four equations that induce the "envelope" shown in Figure 15 [Hissam 2004]. The first equation in the figure computes the average latency for the case where there are no periodic tasks, that is, when the periodic utilization is zero ($U_p=0$). The second equation computes the average latency when the periodic utilization is so high that there is no background processing left. For intermediate levels of periodic utilization, the third and fourth equations compute the average latency for the cases of very small and large periodic periods respectively, thus providing lower and upper bounds for the expected latency of the aperiodic task.
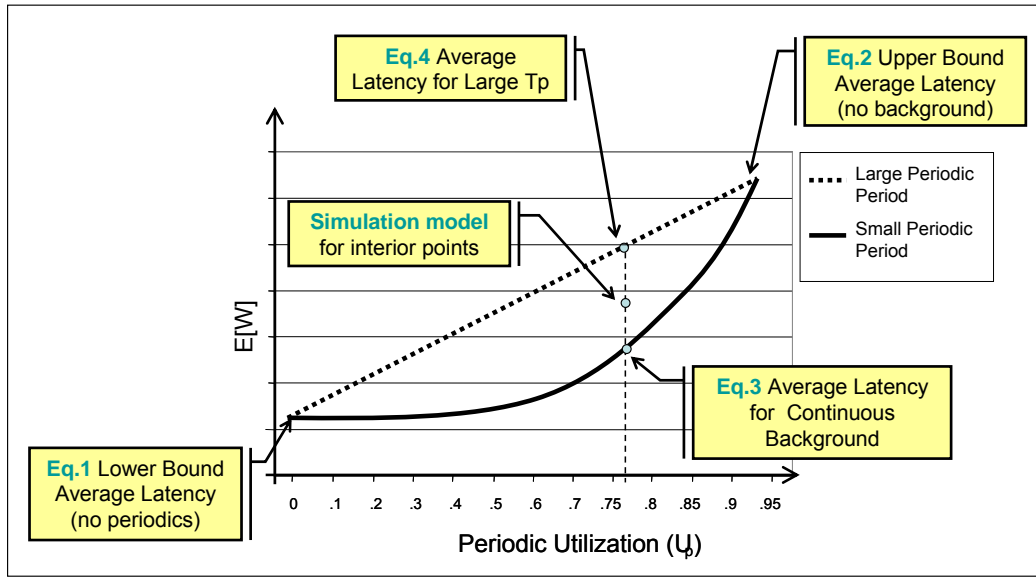
Figure 15:   Lambda-SS Prediction Envelope

If more precision than the bounds provided by this closed-formula evaluation procedure is required, a simulation-based evaluation procedure can be used. This evaluation, which is based on the Extend simulation tool, can simulate the aperiodic task in the sporadic server along with the complete set of periodic tasks in the application. That is, instead of being represented by a single utilization parameter, the periodic task set is modeled including the period and execution time of each of the periodic tasks.

5   Ongoing Work in Lambda-*

In this section we present an overview of the ongoing research we are pursuing to extend the capabilities of the Lambda-* reasoning frameworks.

In many real-time systems there is a need to address a mix of hard- and soft-real-time tasks. For example, hard real-time tasks may include tasks for monitoring power lines and shutting them down when faults are detected, in which missing a deadline can result in catastrophic failure. Firm real-time tasks, such as the data for a frame in a real-time video stream, may still have a deadline (the time at which the frame must be displayed), but do not suffer catastrophic failure in the event of a small number of missed deadlines. For these applications missed deadlines merely correspond to reduced QoS. In a mixed system, we would like to ensure that the hard real-time periodic tasks never miss a deadline while minimizing the miss rate (i.e., the fraction of jobs that miss their deadline) of the firm real-time tasks.

Toward this goal, we have been investigating the application Real-Time Queueing Theory (RTQT) [Doytchinov 2001] to predict the deadline miss ratio of firm real-time tasks in mixed systems. The deadline miss ratio is the fraction of jobs that miss their deadlines, typically expressed as a function of those deadlines. In the context of Lambda-*, we consider two classes of systems:

- static priority systems in which the hard-real-time periodic tasks are assumed to have the highest priority and aperiodic background tasks that run at low priority
- static priority systems with a sporadic server to improve the response times of the aperiodic background tasks

Our research on these classes of systems includes experiments that can be categorized into three types: Two-Flow Static Priority, Two-Flow Sporadic Server, and Multi-Flow. The two-flow static-priority experiments are the simplest of these and set the baseline for expectations in static priority systems. The two-flow sporadic server experiments focus on evaluating the benefit of using a sporadic server on the background task. The multi-flow experiments investigate the effects when there are a large number of periodic tasks and a single background task (or an aggregate of background tasks). In all experiments we assume that the high-priority periodic tasks are scheduled using RMA priorities and never miss deadlines.

5.1   Two Flow Static Priority Experiments

The goal of the two-flow static-priority experiments was to quantify the factors that affect the performance against deadlines of a low-priority aperiodic task sharing the processor with a high-priority task, and to develop a theory for predicting lateness in such systems. Simulation results were compared with two proposed analytical models, an RTQT-based model and a simpler "degraded processor" model in which the low-priority job is considered to be running on a system with a slowed clock. We found that the "degraded processor" model failed to accurately predict lateness of the low-priority task, except when the job size of the high-priority task was very small compared to that of the low-priority task. The RTQT-based formula, on the other hand, fit well over a wide range of conditions. We tested all 16 combinations of conditions in which the inter-

arrival time and service time of the high-priority and low-priority tasks were either exponentially distributed or constant.

The RTQT-based formula we used was

$$\Pr[\text{low-priority task is late}] = e^{-\theta PD} \tag{4}$$

where θ is a workload parameter computed from the mean and standard-deviation of the service time and inter-arrival time for both tasks, P is the fraction of the CPU demand that is due to the low-priority task, and D is the deadline of the low-priority task. In simulations and analysis, we normally consider D to have an arbitrary "unit time" dimension, but if specific units are required, it will be the same as the units used to compute θ. For example, if we use the mean and standard deviation of the inter-arrival and service time in microseconds to compute θ, then D will be the deadline in microseconds. This is because the term θPD is a dimensionless constant with the units for θ and D cancelling each other out.

Figure 16 shows a representative curve from our two-flow experiments showing the deadline miss ratio of a low-priority aperiodic task competing with an aperiodic high-priority task. The high-priority task is assumed to always meet its deadline. Both tasks are assumed to have an exponentially distributed interarrival time with a mean of 2 time units, and an exponentially distributed service time with a mean of 0.95 time units for a total traffic intensity of 0.95. The boxes show the simulation results, the solid line shows the RTQT prediction, and the dashed line shows the degraded processor theory prediction. We can see that while the "degraded processor" theory seriously underpredicts the miss ratio, the RTQT theory provides a close match with the simulation results.
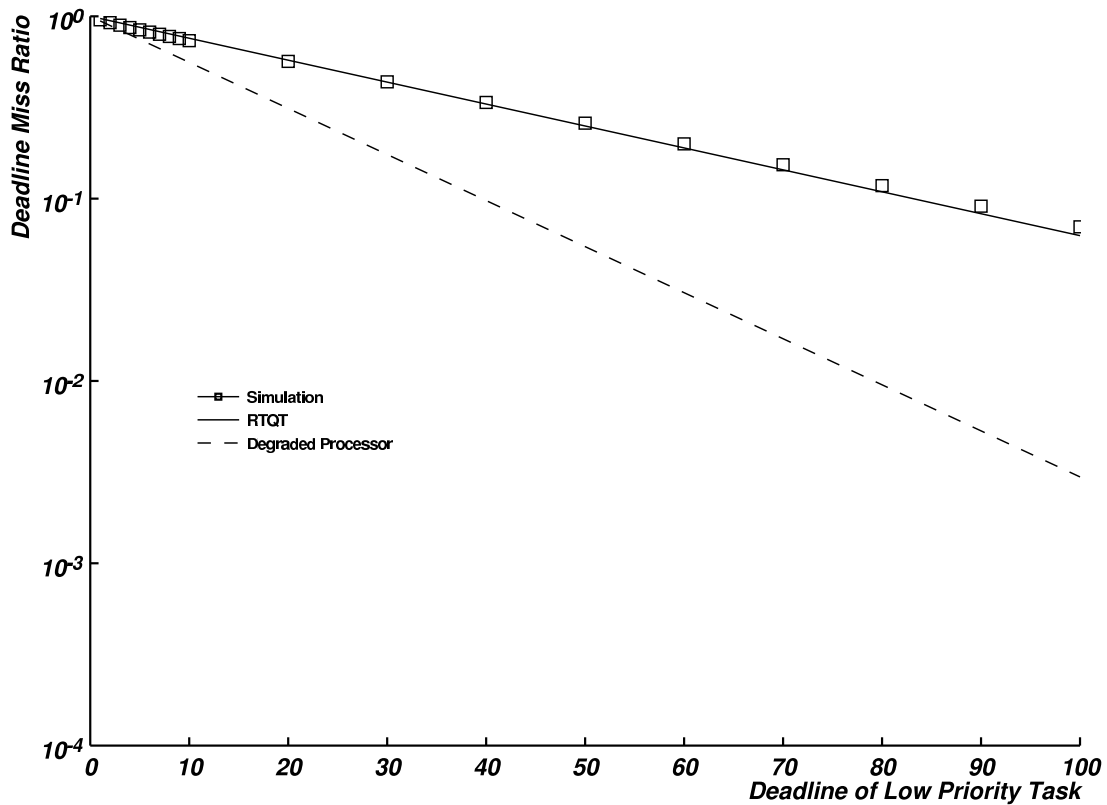
Figure 16:   Comparison of Simulation to Two Proposed Theory Curves

5.2 Two Flow Sporadic Server Experiments

The goal of the two-flow sporadic server experiments was to evaluate the effectiveness of using a sporadic server to decrease lateness of a low-priority aperiodic background task. A sporadic server is characterized by a budget and a replenishment period. As long as budget is available, the aperiodic task can execute at a temporary priority higher than the high-priority task, but when the budget is depleted, it must revert to low priority. The replenishment period controls how long the task must wait for the budget to be replenished.

In these experiments we assumed the following conditions:

- one high-priority periodic task with period P_p and a utilization of 0.45
- one low-priority aperiodic task with a mean-interarrival time of $P_a = 1$ and a utilization of 0.45. All time units in the experiments are relative to the constant $P_a = 1$ value.
- one sporadic server used by the aperiodic task with a replenishment period $P_{ss} = P_p$, and a utilization of 0.55
- total utilization held constant at 0.9

We assume $P_{ss} = P_p$ because we want the sporadic server to have the highest priority but also a budget for as long as possible. In order to ensure the sporadic server has the highest priority, we must choose $P_{ss} \leq P_p$ to satisfy RMA constraints. $P_{ss} = P_p$ will give us the largest possible budget while satisfying the RMA constraints since the budget for a fixed workload level will scale with the period.

Figure 17 shows the results for a representative experiment comparing miss ratio of the aperiodic task with and without a sporadic server (labeled "Sporadic" and "No Sporadic," respectively) as a function of the aperiodic task's deadline. We found that when the period of the high-priority task matched the mean inter-arrival time of the aperiodic task (i.e., $P_p = P_a = 1$), there is no performance difference between using and not using the sporadic server. As the period P_p of the periodic task was increased, the aperiodic miss ratio became higher when no sporadic server was used (curves above the $P_p = 1$ case), and lower when a sporadic server was used (curves below the $P_p = 1$ case).
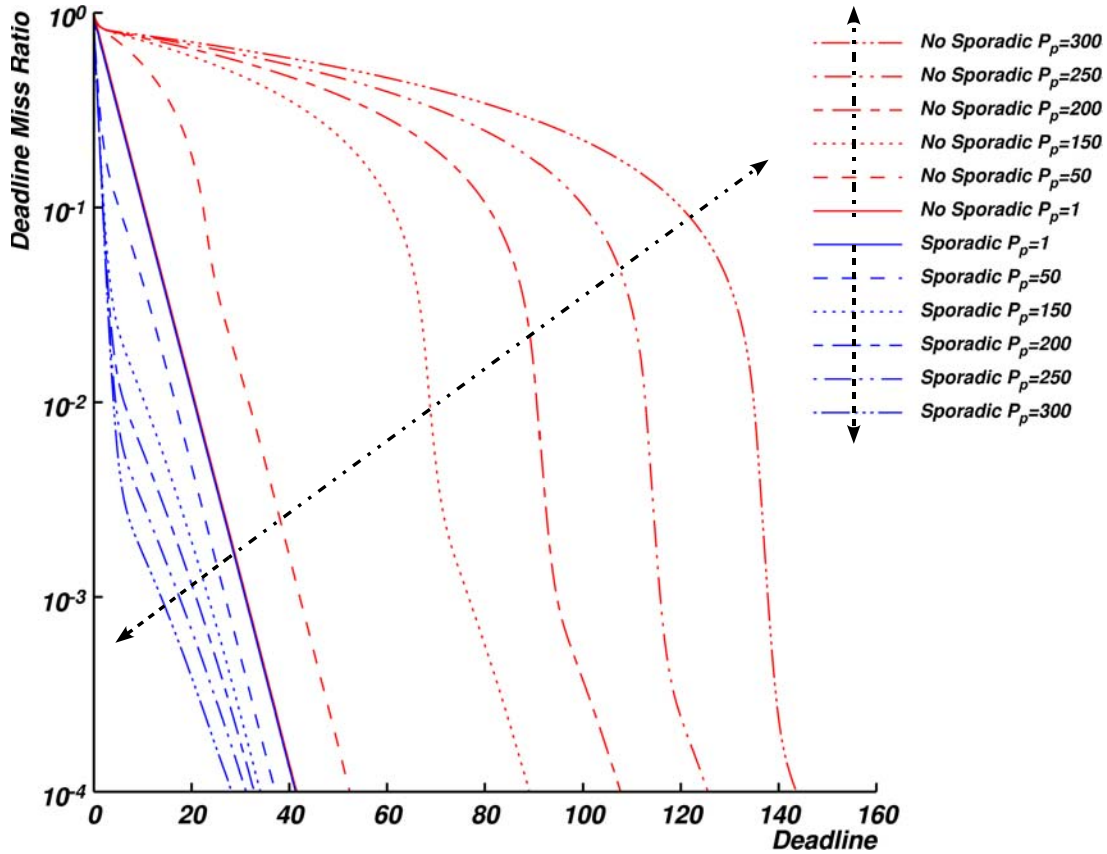
Figure 17: Effect of Sporadic Server on Lateness

The major findings of these experiments were as follows:

- The miss rate of the aperiodic task is minimized when the replenishment period of the sporadic server is as long as possible (i.e., equal to the period of the periodic task). This is because the budget scales with the period, and a larger budget makes it less likely that an aperiodic task will waste budget. Wasted budget can occur when there is not enough remaining budget to complete a job of the aperiodic task. When this happens, the job consumes the budget, but is then forced to wait anyway, resulting in a negligible reduction in latency.

- While holding the utilization of all tasks constant, as the period (P_p) of the high-priority task increases relative to the mean inter-arrival-time of the aperiodic task, the miss rate of the aperiodic task decreases when a sporadic server is used and increases when a sporadic server is not used. The increase in miss rate when the sporadic server is not used is due to the

longer blocking periods causing all aperiodic arrivals to wait longer for the CPU. The decrease in miss rate when the sporadic server is used is likely due to the drop in wasted budget that occurs as the sporadic server replenishment period is increased (which is made possible by the increased period of the high-priority task).

- The sporadic sever is more effective when most of the traffic is periodic. This occurs because for a given total utilization, the demand on the sporadic server goes down as the fraction of traffic that is aperiodic goes down. For example, if the high-priority utilization is 0.8 and the aperiodic utilization is 0.1, the demand (ratio of aperiodic utilization to sporadic server utilization) on the sporadic server will be 0.1/0.2=0.5. On the other hand, if high-priority utilization is 0.1 and the aperiodic utilization is 0.8, the demand on the sporadic server will be 0.8/0.9=0.89. Even though a utilization of 0.9 is available for the sporadic server, there is much higher contention for the server in this case.

5.3 Multi-Flow Experiments

In our multi-flow experiments, we considered cases where there are multiple periodic tasks, and a single aperiodic task (or equivalently, an aggregate of aperiodic tasks). We also considered non-heavy traffic cases. We found that interactions between the periods and computation times of the periodic tasks can have a significant effect on the performance of the background task.

To simplify the problem, we first address the problem of determining lateness for background tasks with zero (or ε) execution time. This decouples the effects of the preemption periods caused by the higher priority periodic tasks, and the queueing effects of the background task. In this simplified case, we found that the miss rate can be expressed by the piece-wise linear equation:

$$\Pr[\text{miss}] = \frac{1}{H}\sum_{i=1}^{N} n_i \max(b_i - D, 0) \tag{5}$$

where H is the length of the hyper-period, N is the number of unique busy period durations occurring over the hyper-period, and n_i and b_i are the number and duration of the busy periods, respectively. Busy periods are periods of time over which the processor is continuously executing some task. The main idea underlying this equation is that the probability of missing a deadline D is the probability of arriving in a busy period with more than D time units until the end of the busy period.

Figure 18 shows this theory curve plotted against a simulation showing near-perfect agreement.
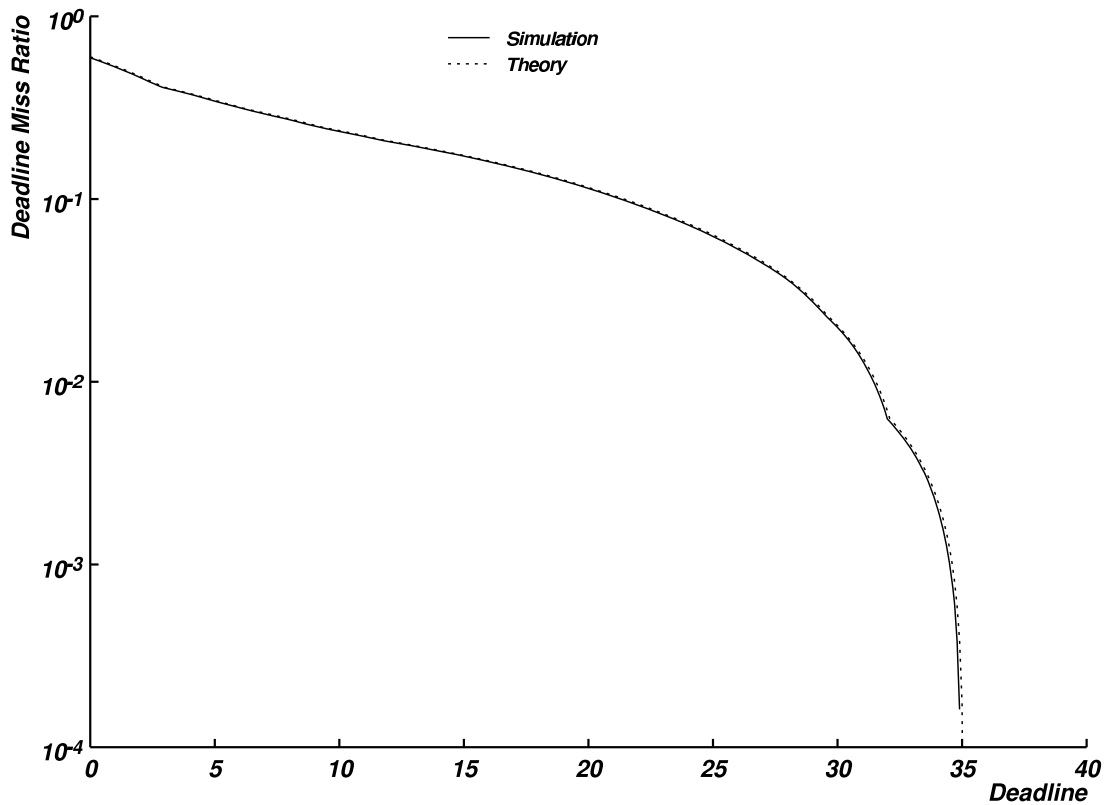
Figure 18:   Multi-Flow Example

The next steps in this work are to relax the zero-execution-time assumption of the background task, and to consider the case with a sporadic server.   As a step in this direction, we have begun investigating the vacation server problem [Doshi 1986]. In a vacation server, the server can randomly enter a vacation which persists for a random period of time. During the vacation, no work is done. Vacation servers can be used to model aperiodic tasks in a system with higher priority periodic tasks where the periodic execution time is considered the "vacation" from the perspective of the low-priority aperiodic. By modeling the busy periods characterized by n_i and b_i as the vacations, we seek to incorporate the queueing delay into Equation (5).

Another application of the vacation server model is as an abstraction of the sporadic server in cases where there is little or no background capacity. In such a case, after the sporadic server exhausts its budget, the aperiodic tasks will be unable to run until the replenishment period elapses. This can be viewed as a sporadic vacation problem where the duration of time in which the budget is exhausted is treated as a vacation.

6 Summary

The Lambda-* reasoning frameworks encapsulate the specialized knowledge of performance analysis for real-time systems in a way that is accessible to software engineering practitioners. Even though reasoning frameworks require adhering to constraints, these restrictions enable predictability. They not only ensure that the assumptions of the underlying analysis theory are satisfied, but also make possible transforming designs into analysis models in a methodical way, which lends itself to automation. The Lambda-* reasoning frameworks automate the creation of the analysis models and their evaluation, two tasks that are very time consuming even for experts in the field.

By supporting worst- and average-case prediction, the Lambda-* reasoning frameworks allow predicting of whether hard and soft timing requirements will be met. In addition, the ongoing research in the use of RTQT to extend Lambda-* aims at predicting the probability of missing a deadline. Such predictions would make it possible to address firm timing requirements without going to the extremes of allocating resources to satisfy worst-case schedulability.

The three reasoning frameworks, Lambda-WBA, Lambda-ABA, and Lambda-SS have been implemented as part of the PACC Starter Kit [Ivers 2008].

Appendix A   Lambda-* Annotations

To make performance predictions, the performance reasoning frameworks require information that is not typically present in a design specification, such as execution and event interarrival times. This information is added to a specification with the CCL annotation mechanism. The annotations supported by the Lambda-* performance reasoning frameworks are described here.

Notation: In the description of the annotations, we use italics to denote placeholders for values or other elements. For example, *pin* is a placeholder for a pin identifier. Note that these identifiers must have the right scoping according to CCL scoping rules. For instance, if component type *Clock* has a pin *tick*, the identifier could be expressed in different ways as shown in these examples, which are not comprehensive.

- `tick` if the annotation is inside the component specification.

- `Clock:tick` if the annotation is outside the component specification and applies to all instances of the component.

- `clock150:tick` if the annotation is inside an assembly specification and applies only to instance *clock150*.

Time units: In annotations that refer to timing, the units are not specified; however, the same unit must be used throughout the model. For example, one can choose to use milliseconds, or microseconds, or another unit of time, as long as all the time expressions are expressed in the same unit.

Statistical Distributions: The Lambda-* performance reasoning frameworks support several statistical distributions for execution and event interarrival time. They are specified as a string with a particular format, as described in Table 2.

Table 2:   Notation for Statistical Distributions in Annotations

Distribution	Format	Description	Example
Constant	C(*value*) C(*value*,*offset*)	A constant *value*. When used as an event interarrival distribution, it can have an *offset*. The offset is the time of the first event arrival.	C(4.5) C(4.5,30)
Uniform	U(*min*,*max*)	Uniform distribution between *min* and *max*	U(3.4,5)
Exponential	M(*mean*)	Exponential distribution with mean *mean*	M(10)
Normal	N(*mean*,*stdDev*)	Normal distribution with mean *mean* and standard deviation *stdDev*	N(120, 6.4)
Generic	G(*min*,*avg*,*max*)	Asymmetric uniform distribution, in which the average is not necessarily in the middle of the [min,max] interval. The distribution consists of two uniformly distributed intervals [min,avg] and (avg,max] with probability proportional to the interval size.	G(3.5,4,8.2)

Execution Time

This annotation indicates the execution time distribution of the computation associated with a sink pin. The execution time is the CPU time the component takes to react to an event on the sink pin

when executing in isolation, that is, without accounting for preemption or blocking on possible interactions through synchronous pins. The value is any legal distribution string; however, some evaluation procedures may not handle certain distributions. Refer to the documentation of each evaluation procedure to determine whether it supports a particular execution time distribution.

It is also possible to use this annotation to indicate the execution time of a service's source pins, since services themselves take time to execute.

Format

```
annotate sinkPin {"lambda*", const string execTime = value }

annotate serviceSourcePin {"lambda*",
                              const string execTime = value }
```

Required Yes

Examples

```
annotate PositionMonitor:input {"lambda*",
              const string execTime = "G(9.99, 10.01, 10.84)" }

annotate oneMonitor:read {"lambda*",
                              const string execTime = "U(3.01, 3.25)" }
```

Execution Overhead

This annotation can be used to add an execution overhead in addition to the execution time indicated for a pin. The value is a float number in the chosen unit of time.

Format

```
annotate sinkPin {"lambda*", const float execOverhead = value }

annotate serviceSourcePin {"lambda*",
                              const float execOverhead = value }
```

Required No

Example

```
annotate PositionMonitor:input {"lambda*",
                                const float execOverhead = 1.15 }
```

Event Interarrival

This annotation indicates the event interarrival distribution of a service's source pin. The value is any legal distribution string; however, some evaluation procedures may not handle certain distributions. Refer to the documentation of each evaluation procedure to determine whether it supports a particular event interarrival distribution.

Periodic events can be denoted by a constant interarrival distribution. However, a special annotation for *period* can be used. The value in this case is the period expressed in units of time.

The offset of a task is the time of the arrival of the first event. The reasoning framework automatically computes the offset of each periodic task to match its offset in the Pin runtime environment.

In Pin, periodic tasks get their first event one period after the start of the assembly. The reasoning framework computes the offset of a task by subtracting the minimum period of all the periodic tasks from the period of the task. The offset can be overridden by specifying it as part of the event interarrival distribution (see the constant distribution with offset). If at least one offset is specified with an annotation, the reasoning framework does not attempt to compute the other offsets. If they have not been specified, they are assumed to be 0.

Format

```
annotate serviceSourcePin {"lambda*",
                            const string eventDistribution = value }
annotate serviceSourcePin {"lambda*", const int period = value }
```

Required Yes

Examples

```
annotate input:data {"lambda*",
                     const string eventDistribution = "M(26.3)" }
annotate clock450:tick {"lambda*", const int period = 450 }
```

Priority

This annotation specifies the priority level of a threaded reaction. The valid range is from 0 to 254, where a higher value indicates a higher priority. This annotation is not strictly a performance annotation, because it is also used for code generation from CCL. However, since it is so important for the performance reasoning framework, it is described here.

Format

```
annotate threadedReaction {"Pin", const int priority = value }
```

Required Yes

Example

```
annotate positionMonitor:reaction {"Pin", const int priority = 15 }
```

Deadline

This annotation can be used to specify a deadline for the response to an event, that is, a deadline for the computation triggered by a service's source pin. By default, the deadline is assumed to be the end of the period in the case of periodic events, and events with random interarrival are assumed not to have a deadline. Those default assumptions can be changed with this annotation. However, since this annotation is not supported by many evaluation procedures, it's advisable to refer to the evaluation procedure documentation to see whether it is supported.

Format

```
annotate serviceSourcePin {"lambda*", const float deadline = value }
```

Required No

Example

```
annotate timer1:go {"lambda*", const float deadline = 325.0 }
```

Connection Overhead

This annotation can be used to account for the connection overhead in a given environment. When used, this connection overhead is considered to be additional execution time preceding the execution of the reaction associated with a sink pin.

Format

```
annotate environment {"lambda*",
                       const float connectionOverhead = value }
```

Required No

Example

```
Rtos env() {

   Rtos:PSClock clk450(450);

};

annotate env { "lambda*", const float connectionOverhead = 100 }
```

Sporadic Server

This annotation specifies the parameters for a component scheduled by a sporadic server. These parameters are

- *budget*: the execution budget of the sporadic server
- *backgroundPriority*: the priority at which the component executes when there is not enough high priority execution budget left in the sporadic server
- *replenishmentPeriod*: the amount of time the sporadic server waits after high priority execution is granted before it replenishes the budget by the previously granted amount

This annotation is not supported by all evaluation procedures.

Format

```
annotate component {"SSContainer",
                     const int backgroundPriority = value,
                     budget = value, replenishmentPeriod = value }
```

Required No

Example

```
annotate trajectoryPlanner {"SSContainer",
                             const int backgroundPriority = 1,
                             budget = 10, replenishmentPeriod = 100 }
```

Scenarios

Scenarios can be used to analyze different operational scenarios in the same design. For instance, a design could have some components that execute only upon a critical condition, and when they do, they will cause other parts of the system to miss their deadlines. In such a case, missing the deadline would be acceptable should the system be attending to a critical condition; however, one must be sure that deadlines are met in other situations. Both situations can be analyzed using the same design by creating scenarios. When the performance reasoning framework is launched, the user selects from the list of scenarios defined in the design to specify which are to be included in the analysis. When the performance model is generated from the design, only those pins belonging to those scenarios are included.

Scenarios are defined by first defining special constants that represent the scenarios, and then indicating which pins belong to which scenarios. The scenario constants are of type `int` and their names must start with the prefix `SCN_`. The value can be any integer 2^i, where $i = 1,...,31$, because the scenarios are considered bit masks.

By default all pins in the assembly participate in all scenarios. If a pin participates only in some scenarios, then this must be specified using an annotation. The annotation can indicate a single scenario or a list of scenarios in which the pin participates.

Format

```
annotate pin {"scenario", const int scenario = value }
annotate pin {"scenario",
              const intArrayType scenario = {value₁,...,valueₙ} }
```

Required No

Example

```
const int SCN_MOV = 1;

const int SCN_PLAN = 2;

const int SCN_MONITOR = 4;

typedef int ScenarioList2[2];

annotate Robot:clock1:tick {"scenario",
                            const ScenarioList2 scenario =
                            {SCN_PLAN, SCN_MOV } }

annotate Robot:clock2:tick {"scenario",
                            const int scenario = SCN_MOV }
```

Concentrator

A concentrator reaction is a reaction that has all of the following characteristics:

- two or more sink pins
- one or more source pins
- only one reaction

- interaction through its source pin(s) every time it has received a message on each of its sink pins.

An example of a concentrator is an adder that has two sink pins, *A* and *B*, for its operands and a source pin *C* to output the result of *A* + *B*. It can produce a result only after it has received one input on each of the sink pins. It can receive a message on *A* first and when it receives a message on *B*, it computes the result and outputs it through *C*. It could also be the other way around.

It is required that the concentrator reaction be the only reaction in the component. Therefore, the term *concentrator* is used interchangeably for reactions and components. However, the annotation refers to a reaction.

The *concentrator* annotation is used to inform the performance reasoning framework that a component behaves as a concentrator. In general, concentrators are non-deterministic: if the inputs are not constrained, it is not possible to determine which input will finally trigger the output source pin. To make predictable which input will trigger, special constraints must be satisfied so that the behavior becomes deterministic. These constraints are

- All the sources of events leading to the concentrator must have the same distribution.
- The concentrator must have one and only one sink for which the highest priority of all its callers is lower than the minimum preceding priority of all the other sinks in the concentrator. This is the pin that will receive the event that will trigger the output of the concentrator.

Format

annotate *reaction* {"lambda*", **const boolean** concentrator = **true** }

Required No

Example

annotate reaction {"lambda*", **const boolean** concentrator = **true** }

Appendix B   Robot Controller Code

This appendix contains the CCL code for the robot controller example used in Section 3. The code for the component and the assembly is included. The specifications of the components do not include the actual internal logic that the components would need to perform their actual functions. Instead, the logic has been replaced with a synthetic load representative of the computation resources that they would require. Nevertheless, the CCL specification for the components and their assembly is sufficiently complete to allow the generation of an implementation that can be run and measured.

File controller.ccl

```
#include "Support/Rtos.ccl"
#include "components/TrajectoryPlanner.ccl"
#include "components/MovementPlanner.ccl"
#include "components/WorkOrderRepository.ccl"
#include "components/AxisControllerSensor.ccl"
#include "components/PositionMonitor.ccl"
#include "components/Sensor.ccl"
#include "components/Monitor.ccl"

assembly Robot () (Rtos)
{
        assume {
                Rtos:PSClock clock130(130);
                Rtos:PSClock clock450(450);
                Rtos:PSClock clock150(150);
                Rtos:PSClock clock2000(2000);
        }

        TrajectoryPlanner trajectoryPlanner();
        MovementPlanner movementPlanner();
        WorkOrderRepository repository();
        AxisController controllerX("X");
        AxisController controllerY("Y");
        Sensor sensor();
        PositionMonitor positionMonitor();
        Monitor monitor();

        clock450:tick ~> trajectoryPlanner:go;
        trajectoryPlanner:put ~> repository:access;

        clock150:tick ~> movementPlanner:go;
        movementPlanner:get ~> repository:access;
        movementPlanner:moveX ~> controllerX:move;
        movementPlanner:moveY ~> controllerY:move;
        controllerX:position ~> positionMonitor:input;
        controllerY:position ~> positionMonitor:input;

        clock130:tick ~> sensor:go;
        sensor:position ~> positionMonitor:input;

        trajectoryPlanner:read ~> positionMonitor:read;

        clock2000:tick ~> monitor:go;

        // priorities
        annotate Rtos:PSClock:reaction {"Pin", const int priority = 254 }
        annotate trajectoryPlanner:reaction {"Pin", const int priority = 4}
        annotate movementPlanner:reaction {"Pin", const int priority = 16}
```

```
            annotate repository:reaction {"Pin", const int priority = 18}
            annotate controllerX:reaction {"Pin", const int priority = 20}
            annotate controllerY:reaction {"Pin", const int priority = 20}
            annotate sensor:reaction {"Pin", const int priority = 10 }
            annotate PositionMonitor:inputReaction {"Pin", const int priority = 12 }
            annotate PositionMonitor:readReaction {"Pin", const int priority = 14 }
            annotate monitor:reaction {"Pin", const int priority = 2 }

            annotate TrajectoryPlanner:reaction {"Pin", const int queueLength = 1}
            annotate MovementPlanner:reaction {"Pin", const int queueLength = 1}
            annotate Sensor:reaction {"Pin", const int queueLength = 1}

            expose {}
}

Rtos env() {
            Rtos:PSClock clk130(130);
            Rtos:PSClock clk450(450);
            Rtos:PSClock clk150(150);
            Rtos:PSClock clk2000(2000);
};

Robot robot() {
            Robot:clock130 = env:clk130;
            Robot:clock450 = env:clk450;
            Robot:clock150 = env:clk150;
            Robot:clock2000 = env:clk2000;
};

const int SCN_MOV = 1;
const int SCN_PLAN = 2;
annotate Robot:clock450:tick { "scenario", const int scenario = SCN_PLAN }
annotate Robot:clock150:tick { "scenario", const int scenario = SCN_MOV }

// lambda* annotations
annotate Robot:clock130:tick {"lambda*", const string eventDistribution = "C(130)" }
annotate Robot:clock450:tick {"lambda*", const string eventDistribution = "C(450)" }
annotate Robot:clock150:tick {"lambda*", const string eventDistribution = "C(150)" }
annotate Robot:clock2000:tick {"lambda*", const string eventDistribution = "C(2000)" }

annotate Robot:clock450:tick {"lambda*", const float deadline = 450.0 }
annotate Robot:clock150:tick {"lambda*", const float deadline = 150.0 }
```

File components/TrajectoryPlanner.ccl

```
#include "Support/SyntheticLoad.ccl"

component TrajectoryPlanner() {
            sink asynch go();
            source synch read();
            source synch put(produce int mode, produce string in, consume string out);

            threaded react reaction (go, read, put) {
                    start -> listen {}
                    listen -> getPosition {
                            trigger ^go;
                            action {
                                    syntheticLoad(90);
                                    ^read();
                            }
                    }
                    getPosition -> store {
                            trigger $read;
                            action ^put(1, "there");
                    }
                    store -> listen {
```

```
                            trigger $put;
                            action $go();
                    }
            }
    }

annotate TrajectoryPlanner:go {"lambda*",
        const string execTime =  "G(88.5, 89.5, 90.5)" }
```

File components/MovementPlanner.ccl

```
#include "Support/SyntheticLoad.ccl"

component MovementPlanner() {
        sink asynch go();
        source synch get(produce int mode, produce string in, consume string out);
        source asynch moveX(produce int pos);
        source asynch moveY(produce int pos);

        threaded react reaction (go, get, moveX, moveY) {
                string where;
                start -> listen {}
                listen -> retrieve {
                        trigger ^go;
                        action {
                                syntheticLoad(20);
                                ^get(0, "");
                        }
                }
                retrieve -> controlX {
                        trigger $get;
                        action {
                                where = get.out;
                                //%{ Printf("Moving to: %s\n", $ccl$where); %}
                                ^moveX(10);
                        }
                }
                controlX -> controlY {
                        trigger $moveX;
                        action ^moveY(10);
                }
                controlY -> listen {
                        trigger $moveY;
                        action $go();
                }
        }
}

annotate MovementPlanner:go {"lambda*",
        const string execTime =  "G(18.8, 20.0, 21.00)" }
```

File components/WorkOrderRepository.ccl

```
#include "Support/SyntheticLoad.ccl"

component WorkOrderRepository() {
        sink synch access(consume int mode, consume string in, produce string out);

        threaded react reaction (access) {
                int items = 2;
                start -> listen {}
                listen -> putting {
                        trigger ^access;
                        guard access.mode == 1;
                }
```

```
                  listen -> getting {
                          trigger ^access;
                          guard access.mode == 0;
                  }
                  putting -> listen {
                          action {
                                  items = items + 3;
                                  //%{ Printf("\tPutting: remaining items %d\n", $ccl$items); %}
                                  syntheticLoad(20);
                                  $access("");
                          }
                  }
                  getting -> listen {
                          guard items > 0;
                          action {
                                  syntheticLoad(20);
                                  //%{ Printf("\tremaining items %d\n", $ccl$items); %}
                                  items = items - 1;
                                  $access("there");
                          }
                  }
                  getting -> listen {
                          guard items <= 0;
                          action {
                                  %{ Printf("\n\n\tABORTING....no movement data to drive robot\n\n\n");
%}
                                  alert(2, "Aborting...work order repository is empty");
                                  $access("n/a");
                          }
                  }
          }
}

annotate WorkOrderRepository:access {"lambda*",
        const string execTime =  "G(19.8, 19.9, 20.8)" }
```

File components/AxisControllerSensor.ccl

```
#include "Support/SyntheticLoad.ccl"

component AxisController(string name) {
        sink asynch move(consume int pos);
        source asynch position();

        threaded react reaction (move, position) {
                start -> listen {}
                listen -> moving {
                        trigger ^move;
                        action {
                                //%{ Printf("\tmoving axis %s\n", $ccl$name); %}
                                syntheticLoad(13);
                        }
                }
                moving -> sending {
                        action {
                                $move();
                                ^position();
                        }
                }
                sending -> listen {
                        trigger $position;
                }
        }
}
```

```
annotate AxisController:move {"lambda*",
          const string execTime =  "G(12.8, 13.0, 13.5)" }
```

File components/PositionMonitor.ccl
```
#include "Support/SyntheticLoad.ccl"

component PositionMonitor() {
          sink asynch input();
          sink synch read();

          threaded react inputReaction (input) {
                    start -> listen {}
                    listen -> listen {
                              trigger ^input;
                              action {
                                        syntheticLoad(10);
                                        $input();
                              }
                    }
          }

          threaded react readReaction (read) {
                    start -> listen {}
                    listen -> listen {
                              trigger ^read;
                              action {
                                        syntheticLoad(3);
                                        $read();
                              }
                    }
          }
}

annotate PositionMonitor:input {"lambda*",
          const string execTime =  "G(9.8, 10.0, 10.8)"  }

annotate PositionMonitor:read {"lambda*",
          const string execTime =  "G(3.0, 3.1, 3.2)"  }
```

File components/Sensor.ccl
```
#include "Support/SyntheticLoad.ccl"

component Sensor() {
          sink asynch go();
          source asynch position();

          threaded react reaction (go, position) {
                    start -> listen {}
                    listen -> sense {
                              trigger ^go;
                              action {
                                        syntheticLoad(5);
                                        ^position();
                              }
                    }
                    sense -> listen {
                              trigger $position;
                              action $go();
                    }
          }
```

```
}

annotate Sensor:go {"lambda*",
        const string execTime =  "G(5.0, 5.1, 5.6)" }




File components/Monitor.ccl
component Monitor() {
        sink asynch go();

        threaded react reaction (go) {
                start -> listen {}
                listen -> listen {
                        trigger ^go;
                        action {
                                %{ Printf("\nRobot working...\n"); %}
                                $go();
                        }
                }
        }
}

annotate Monitor:go {"lambda*",
        const string execTime =  "G(0.25, 0.3, 0.5)" }
```

References/Bibliography

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

[Bass 2005]

Bass, Len; Ivers, James; Klein, Mark; & Merson, Paulo. *Reasoning Frameworks* (CMU/SEI-2005-TR-007, ADA441248). Software Engineering Institute, Carnegie Mellon University, 2005. http://www.sei.cmu.edu/pub/documents/05.reports/pdf/05tr007.pdf

[Bernat 2001]

Bernat, G.; Burns, A.; & Llamos, A. "Weakly Hard Real-Time Systems." *IEEE Transactions on Computers 50*, 4 (April 2001):308-321.

[Buttazzo 1999]

Buttazzo, G. C. & Caccamo, M. "Minimizing Aperiodic Response Times In A Firm Real-Time Environment." *IEEE Transactions on Software Engineering 25*, 1 (Jan/Feb 1999): 22-32.

[Clements 2003]

Clements, Paul. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.

[Davari 1992]

Davari, S. & Sha, L. "Sources of Unbounded Priority Inversions in Real-Time Systems and a Comparative Study of Possible Solutions." *SIGOPS Operating Systems Review*. 26, 2 (Apr. 1992): 110-120.

[Diaz-Pace 2008]

Diaz-Pace, Andres; Kim, Hyunwoo; Bass, Len; Bianco, Philip; & Bachmann, Felix. "Integrating Quality Attribute Reasoning Frameworks in the ArchE Design Assistant." *Quality of Software Architecture: Models and Architecture, Volume 5281*. Springer Berlin, 2008.

[Doshi 1986]

Doshi, B. T. "Queueing Systems with Vacations—a Survey." *Queueing Systems 1*, 1 (June 1986): 29-66.

[Doytchinov 2001]

Doytchinov, B.; Lehoczky, J. P.; & Shreve S. "Real-Time Queues in Heavy Traffic with Earliest-Deadline-First Queue Discipline." *Annals of Applied Probability 11*, 2 (May 2001): 332-378.

[Gonzalez Harbour 1991]

Gonzalez Harbour, M.; Klein, M.; & Lehoczky, J. "Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority," 116-128. *Proceedings of IEEE Real-Time Systems Symposium*. San Antonio, Texas, December 1991. IEEE Computer Society Press, 1991.

[Gonzalez Harbour 2001]

Gonzalez Harbour, M.; Gutierrez Garcia, J.J.; Palencia Gutierrez, J.C.; Drake Moyano, J. M. "MAST: Modeling and Analysis Suite for Real-Time Applications," 125. *Proceedings of 13th*

*Euromicro Conference on Real-Time Systems (ECRTS).*Washington, DC, June 2001. IEEE Computer Society, 2001.

[Hissam 2002]

Hissam, Scott; Hudak, John; Ivers, James; Klein, Mark; Larsson, Magnus; Moreno, Gabriel; Northrop, Linda; Plakosh, Daniel; Stafford, Judith; Wallnau, Kurt; & Wood, William. *Predictable Assembly of Substation Automation Systems: An Experiment Report, Second Edition* (CMU/SEI-2002-TR-031, ADA418441). Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr031.pdf

[Hissam 2004]

Hissam, Scott; Klein, Mark; Lehoczky, John; Merson, Paulo; Moreno, Gabriel; & Wallnau, Kurt. *Performance Property Theories for Predictable Assembly from Certifiable Components (PACC)* (CMU/SEI-2004-TR-017, ADA431163). Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/pub/documents/04.reports/pdf/04tr017.pdf

[Hissam 2005a]

Hissam, Scott; Ivers, James; Plakosh, Daniel; & Wallnau, Kurt. *Pin Component Technology (V1.0) and Its C Interface* (CMU/SEI-2005-TN-001, ADA441815). Software Engineering Institute, Carnegie Mellon University, 2005.
http://www.sei.cmu.edu/pub/documents/05.reports/pdf/05tn001.pdf

[Hissam 2005b]

Hissam, Scott A.; Moreno, Gabriel A.; &. Wallnau, Kurt C. *Using Containers to Enforce Smart Constraints for Performance in Industrial Systems* (CMU/SEI-2005-TN-040, ADA445164). Software Engineering Institute, Carnegie Mellon University, 2005.
http://www.sei.cmu.edu/pub/documents/05.reports/pdf/05tn040.pdf

[Hissam 2008]

Hissam, S.; Moreno, G.; Plakosh, D.; Savo, I.; & Stelmarczyk, M. "Predicting the Behavior of a Highly Configurable Component Based Real-Time System," 57-68. *Proceedings of the 20th Euromicro Conference on Real-Time Systems*. Prague, Czech Republic, July 2008. IEEE Computer Society, 2008.

[Ivers 2004]

Ivers, James & Sharygina, Natasha. *Overview of ComFoRT: A Model Checking Reasoning Framework* (CMU/SEI-2004-TN-018, ADA442864). Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/pub/documents/04.reports/pdf/04tn018.pdf

[Ivers 2007]

Ivers, J. &. Moreno, G. "Model-driven Development with Predictable Quality," 874-875. *Proceedings of International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '07)*. Montreal, Quebec, October 2007. Association for Computer Machinery, 2007.

[Ivers 2008]

Ivers, J. & Moreno, G. "PACC Starter Kit: Developing Software with Predictable Behavior," 949-950. *ICSE Companion '08: Companion of the 30th International Conference on Software Engineering.* Leipzig, Germany, May 2008. Association for Computer Machinery, 2008.

[Klein 1993]

Klein, Mark; Ralya, Tom; Pollak, Bill; & Obenza, Ray. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Kluwer Academic Press, 1993.

[Kleinrock 1975]

Kleinrock, L. *Queueing Systems Volume 1: Theory.* Wiley Interscience, 1975.

[Krahl 2001]

Krahl, D. "Extend: the Extend Simulation Environment," 217-225. *WSC '02: Proceedings of the 34th Winter Simulation Conference.* Arlington, VA, December 2001. IEEE Computer Society, 2001.

[Laplante 2004]

Laplante, Phillip A. *Real-Time Systems Design and Analysis, 3rd Edition.* Wiley-IEEE Press, 2004.

[Larsson 2004]

Larsson, M. "Predicting Quality Attributes in Component-Based Software Systems." PhD diss., Mälardalen University, Sweden, 2004.

[Lopez Martinez 2004]

Lopez Martinez, Patricia; Medina, Julio Luis; & Drake, Jose M. "Sim-MAST: Simulador de Sistemas Distribuidos de Tiempo Real." *XII Jornadas de Concurrencia y Sistemas Distribuidos*, 2004.

[Moreno 2002]

Moreno, G.; Hissam, S.; & Wallnau, K. "Statistical Models for Empirical Component Properties and Assembly-Level Property Predictions: Toward Standard Labeling." *Proceedings of the 5th International Workshop on Component-Based Software Engineering*, in conjunction with the *24th International Conference on Software Engineering (ICSE2002).* Orlando, FL, May 2002. IEEE Computer Society, 2002.

[Moreno 2006]

Moreno, G. "Creating Custom Containers with Generative Techniques," 29-38. *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06).* Portland, OR, October 2006. Association for Computer Machinery, 2006.

[Moreno 2008]

Moreno, Gabriel; Smith, Connie; & Williams, Lloyd. "Performance Analysis of Real-Time Component Architectures: A Model Interchange Approach," 115-126. *7th International Workshop on Software and Performance*, Princeton, NJ, June 2008. Association for Computer Machinery, 2008.

[Sha 1990]

Sha, Lui & Goodenough, J. B. "Real-Time Scheduling Theory and Ada." *Computer*, *23*, 4 (April 1990): 53-62.

[Sprunt 1989]

Sprunt, Brinkley & Sha, Lui. *Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System* (CMU/SEI-89-TR-011, ADA211344). Software Engineering Institute, Carnegie Mellon University, 1989. http://www.sei.cmu.edu/pub/documents/89.reports/pdf/tr11.89.pdf

[Wallnau 2003]

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

[Woodside 2007]

Woodside, C. M.; Franks, G.; & Petriu, D. C. "The Future of Software Performance Engineering," 171-187. *29th International Conference on Software Engineering (ICSE)*. Washington, DC, May 2007. IEEE Computer Society, 2007.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE February 2009	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE
Overview of the Lambda-* Performance Reasoning Frameworks

5. FUNDING NUMBERS
FA8721-05-C-0003

6. AUTHOR(S)
Gabriel A. Moreno and Jeffrey Hansen

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

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-2008-TR-020

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
ESC-TR-2008-020

11. SUPPLEMENTARY NOTES

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

12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

The Predictable Assembly from Certifiable Code (PACC) Initiative at the Carnegie Mellon Software Engineering Institute is developing methods and technologies to enable the production of software with predictable behavior by making the application of analytic methods accessible to software engineering practitioners. The use of reasoning frameworks is a means to achieving this goal. A reasoning framework is a packaging of an analysis theory along with other important elements that are needed for its application, such as methods for creating analysis models and evaluating them.

Lambda-* is a suite of performance reasoning frameworks founded on the principles of Generalized Rate Monotonic Analysis (GRMA) for predicting the average and worst-case latency of periodic and stochastic tasks in real-time systems. Lambda-* can be applied to many different, uniprocessor, real-time systems having a mix of tasks with hard and soft deadlines with periodic and stochastic event in-terarrivals. Some examples include embedded control systems (e.g., avionic, automotive, robotic) and multimedia systems (e.g., audio mixing).

This report provides an overview of the Lambda-* performance reasoning frameworks, their current capabilities, and ongoing research. The Lambda-* reasoning frameworks have been implemented as a part of the PACC Starter Kit (PSK), a development environment that integrates a collection of technologies to enable the development of software with predictable runtime behavior.

14. SUBJECT TERMS
performance, real-time, reasoning framework, predictable assembly

15. NUMBER OF PAGES
66

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