Reasoning Frameworks

Len Bass
James Ivers
Mark Klein
Paulo Merson

July 2005

TECHNICAL REPORT
CMU/SEI-2005-TR-007
ESC-TR-2005-007

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Reasoning Frameworks

CMU/SEI-2005-TR-007
ESC-TR-2005-007

Len Bass
James Ivers
Mark Klein
Paulo Merson

July 2005

Software Architecture Technology Initiative
Predictable Assembly from Certifiable Components Initiative

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Christos Scondras
Chief of Programs, XPK

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

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

Table of Contents

List of Figures

List of Tables

Abstract

Determining whether a system will satisfy critical quality attribute requirements in areas such as performance, modifiability, and reliability is a complicated task that often requires the use of many complex theories and tools to arrive at reliable answers. This report describes a vehicle for encapsulating the quality attribute knowledge needed to understand a system's quality behavior as a reasoning framework that can be used by nonexperts. A reasoning framework includes the mechanisms needed to use sound analytic theories to analyze the behavior of a system with respect to some quality attribute. This report defines the elements of a reasoning framework and illustrates the reasoning framework concept by describing several reasoning frameworks and how they realize these elements.

1 Introduction

A system's software architecture has a significant impact on its ability to satisfy critical quality attribute requirements in areas such as performance, modifiability, and reliability. Determining whether an architecture will satisfy quality requirements is a complicated task, however, often requiring the use of many complex theories and tools to arrive at reliable answers.

This report describes a vehicle for encapsulating the quality attribute knowledge needed to understand a system's quality behavior as a reasoning framework that can be used by nonexperts. A reasoning framework includes the mechanisms needed to use sound analytic theories to analyze the behavior of a system with respect to some quality attribute. For example, a performance reasoning framework could generate a task model corresponding to an architectural description and then calculate latencies using Rate Monotonic Analysis (RMA). The reasoning ability given by the theories serves as a basis for (*a*) predicting behavior before the system is built, (*b*) understanding behavior after it is built, and (*c*) making design decisions while it is being built and when it evolves.

Each reasoning framework is based on specific analytic theories that are used to compute specific measures of quality attribute related behavior. This explicit scoping of the applicability of a reasoning framework leads to analytic power, often in the form of improved confidence in analytic results or improved analytic tractability. However, this scoping does imply a need to apply multiple reasoning frameworks to understand a system's behavior for multiple quality attributes.

Having said this, however, reasoning frameworks were designed to be independent of the development method or even time of application. Though our current work, and the remainder of this report, focuses on the application to architecture descriptions, reasoning frameworks could be applied to a variety of development artifacts such as designs, source code, or even binary images. Reasoning frameworks could be applied to software architectures to help guide initial decisions or to source code as a means to justify reduced testing effort. Regardless, the essence—packaging the knowledge needed to analyze quality behaviors—is the same.

In our work, we use reasoning frameworks as part of two different technologies: (1) ArchE and (2) prediction-enabled component technologies (PECTs). ArchE is an architectural design assistant that uses reasoning frameworks to analyze quality behavior and that adjusts designs using architectural tactics to satisfy unmet architectural requirements [Bachmann 03]. PECTs combine reasoning frameworks that predict system behaviors with a component technology that constructively enforces applicability of the reasoning frameworks [Wallnau 03a].

Section 2 of this report defines the reasoning framework concept by describing the elements that make up a reasoning framework and how they interact to help engineers reason about quality attributes. Section 3 describes related techniques that can or should be used along with reasoning frameworks. Section 4 describes a reasoning framework that we have developed and how it realizes the elements of our definition, to make the idea more concrete.

2 Elements of a Reasoning Framework

A reasoning framework provides the capability to reason about specific quality attribute behavior(s) of an architecture through the use of analytic theories. The class of behaviors or problem situations for which the reasoning framework is useful is referred to as the *problem description*. Each reasoning framework makes use of an *analytic theory* as the basis for its reasoning. Analytic theories use formal *model representations* to abstractly describe those aspects of a system about which they reason. The constraints imposed by the analytic theory on the architecture, that is, the *analytic constraints*, ensure that the assumptions for using the theory are met. The mapping from the architecture to the model is known as the *interpretation*, and the procedure used to solve the model is known as an *evaluation procedure*. In summary, the six elements of a reasoning framework are

1. problem description: identifies the quality attribute for which the reasoning framework is used, more specifically the set of quality measures that can be calculated or the set of quality requirements (as constrained by quality measures) whose satisfaction it evaluates
2. analytic theory: the sound foundations on which analyses are based; typically an established discipline such as queuing theory, rate monotonic scheduling theory, finite state automata, or temporal logic
3. analytic constraints: constraints specifying the set of systems that satisfy assumptions imposed by the analytic theory
4. model representation: a model of the aspects of a system relevant to the analytic theory in a form suitable for use with the evaluation procedure
5. interpretation: a procedure that generates model representations from information found in architectural descriptions
6. evaluation procedure: algorithms or formulae that calculate specific measures of a quality attribute from a model representation

These elements encapsulate the quality attribute knowledge needed to predict some aspect of a system’s behavior with respect to some quality attribute. The specific behavior predicted is given by the quality attribute measures of interest in the reasoning framework’s problem description. Because the six elements have a well-defined scope in terms of quality attributes, a reasoning framework contains only what is necessary to reason about the specified quality behaviors.

While all elements are essential to form a coherent reasoning framework, not all elements have runtime manifestations (i.e., not all elements are found in a reasoning framework implementation). Nor are all elements visible to all stakeholders. Reasoning framework developers can be expected to be quality attribute and analytic theory experts. However, reasoning framework users can be nonexperts because of the way quality attribute knowledge is packaged into reasoning frameworks. Users don't need to be experts in the analytic theory and don't need to understand the model representation or how interpretation or evaluation procedures work—they are interested in the final result. The implementation of a reasoning framework, in fact, should be as automated as possible and should expose users to as few concepts as possible. It is important to note, however, that while a reasoning framework does not have to provide an implementation, it must be defined rigorously enough to be implementable.[1] It particular, the evaluation procedure must be computable.

From an end user's point of view, the reasoning framework is a black box that takes an architecture description and desired quality attribute measures as input and produces quality attribute measures as output (Figure 1). For the user, the breakdown of reasoning framework elements and the underlying analytic theory are less important than how the framework is used in practice.

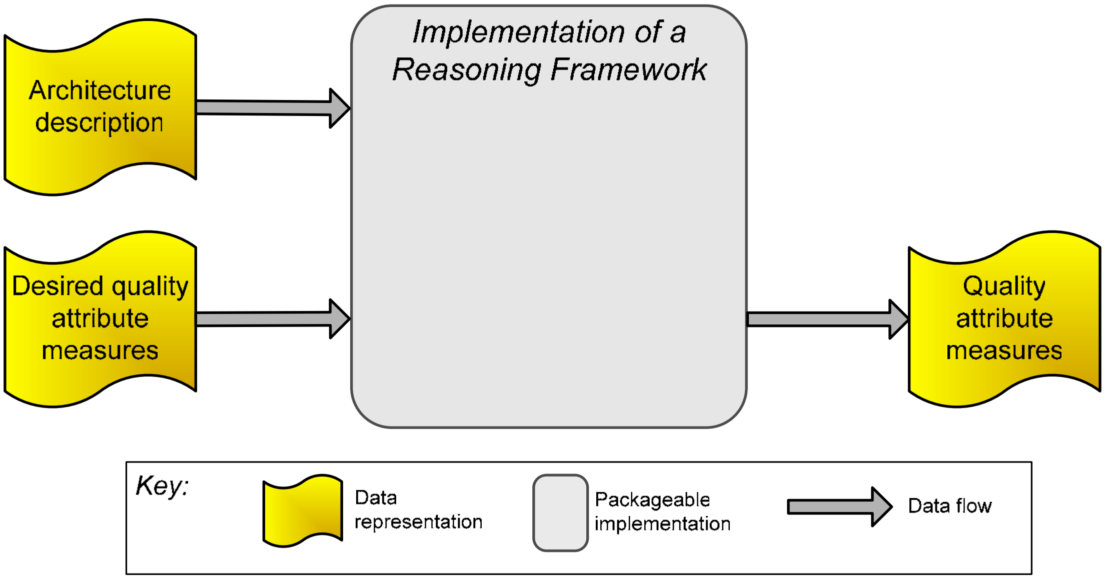
Implementation of a Reasoning Framework
Architecture description
Desired quality attribute measures
Quality attribute measures
Key:
Data representation
Packageable implementation
Data flow

Figure 1: A Reasoning Framework from a User's Perspective

Figure 2 shows what the implementation of a reasoning framework might look like. The architecture description provided by the user as input must satisfy the reasoning framework's analytic constraints. The set of desired quality attribute measures is restricted to problems from the reasoning framework's problem description. Inside, two transformations take place. First, the architecture description is translated into a model representation via interpretation. Second, the evaluation procedure reads the model representation (and sometimes the desired

[1] Of course, only those elements with a runtime manifestation are implemented (or must be implementable). Such elements include the evaluation procedure and interpretation but not the analytic theory.

quality attribute measures) to predict the quality attribute measures that are the output of the reasoning framework. The evaluation procedure implementation uses algorithms and formulae based on the analytic theory.

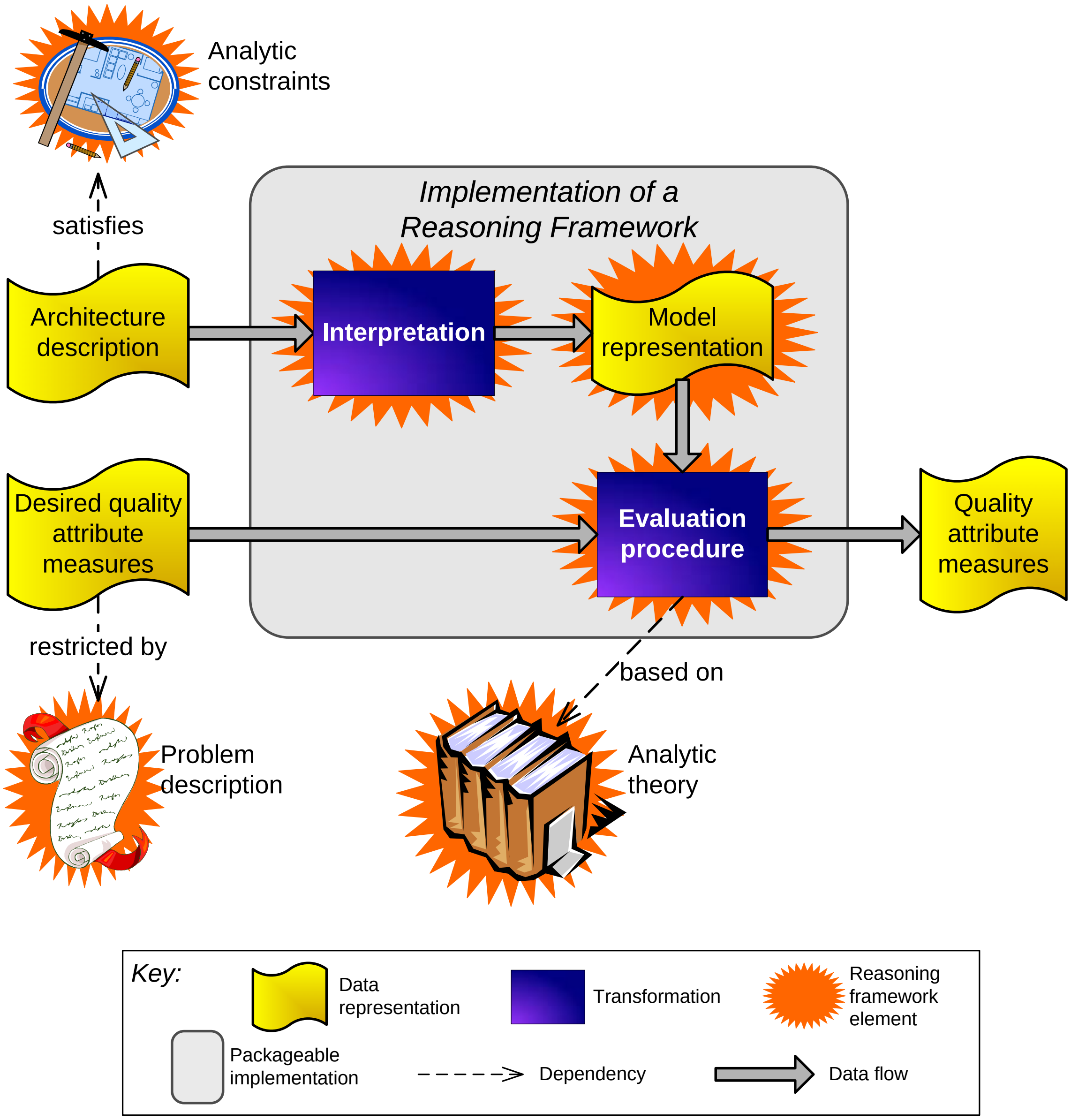

Figure 2: Implementation of a Reasoning Framework

In Sections 2.1 through 2.6, we describe the six elements of a reasoning framework in more detail. Additionally, we briefly illustrate how each element is realized in the following three reasoning frameworks that we use as running examples:

1. a performance reasoning framework [Hissam 02]
2. a variability reasoning framework[2]
3. a model checking reasoning framework (ComFoRT), which is further described in Section 4 [Ivers 04]

In Section 2.7 we provide a table summarizing the six elements of our three reasoning framework examples.

2.1 Problem Description

A reasoning framework is designed to solve specific quality attribute problems; its problem description describes this set of problems, allowing users to choose reasoning frameworks that fit their needs. The problem description identifies the quality attribute for which the reasoning framework is used, more specifically the set of quality measures that can be calculated or the set of quality requirements (as constrained by quality measures) whose satisfaction it evaluates. For example, while performance is a broad topic with many potential measures, a particular performance reasoning framework might only calculate task latency or evaluate whether tasks will meet their deadlines.

The problem description also describes the notation that must be used to express the desired quality measures or quality requirements, as they relate to a user's architecture description. Continuing the above example, the problem description might dictate that a user identify the tasks for which latency should be calculated by annotating the appropriate elements of the architecture description with a particular tag.

An effective system for describing the problems that can be solved by a reasoning framework is the general scenario approach [Bass 03]. A *general scenario* is a system-independent description of a quality attribute requirement that can be used for *any* quality attribute. A general scenario has six parts:

1. *stimulus*: a condition that needs to be considered when it arrives at a system. Examples are an event arriving at an executing system and a change request arriving at a developer.
2. *source of the stimulus*: the entity (e.g., a human or computer system) that generated the stimulus. For example, a request arriving from a trusted user of the system will in some circumstances be treated differently than a request arriving from an untrusted user.
3. *environment*: the conditions under which the stimulus occurs. For example, an event that arrives when the system is in an overload condition may be treated differently than an event that arrives when the system is operating normally.
4. *artifact*: the artifact affected by the stimulus. Examples include the system, the network, and a subsystem.

[2] Bachmann, F. *Variability Reasoning Framework for Product Line Architectures* (CMU/SEI-2005-TR-012, to be published in 2005).

5. *response*: the activity undertaken by the artifact after the arrival of the stimulus. Examples include processing the event for event arrival or making the change without affecting other functionality for a change request.
6. *response measure*: the attribute-specific constraint that must be satisfied by the response. A constraint could be, for example, that the event must be processed within 100 ms, or that the change must be made within two person-days. Response measures can also be Boolean, such as “The user has the ability to cancel a particular command” (yes or no).

Software Architecture in Practice [Bass 03] provides a set of general-scenario-generation tables for the attributes of availability, modifiability, performance, security, testability, and usability. These tables can be used to elicit the specific quality attribute requirements for their particular attributes. A reasoning framework’s problem description can consist of the set of general scenarios for which it can calculate the response measure.

Concrete scenarios are instantiations of general scenarios for the particular system under consideration. These can likewise be used to specify the quality requirements (or desired measures) for that system. The set of concrete scenarios supplied by a user indicates the specific problems the user wants to use the reasoning framework to analyze and solve.

The following is a sample concrete scenario: A remote user initiates a set of transactions to the system aperiodically no more often than one per second under normal operations. The transactions are to be processed with an average latency of two seconds.

Table 1 summarizes the problem descriptions of the three reasoning frameworks introduced in Section 2.

Table 1: Problem Descriptions of Three Reasoning Frameworks

Reasoning Framework	**Problem Description**
Performance	prediction of average latency
Variability	prediction of the cost of creating a new variant
ComFoRT	prediction of satisfaction of behavioral assertions expressed in state/event linear temporal logic (SE-LTL)

2.2 Analytic Theory

Each reasoning framework is based on one or more analytic theories. An analytic theory[3] is some body of knowledge that has proven to be useful in reasoning about some quality attribute. Analytic theories are drawn from established computer science disciplines and include topics such as RMA, automata theory, queuing theory, and Markov analysis. An analytic theory defines its assumptions, the elements and their properties describable in the theory, rules

[3] In other reports, this element has been referred to as quality attribute theory or property theory.

governing the relationships among elements, and a logic for drawing conclusions from a specific model in the theory.

For example, RMA theory is used to reason about worst-case latency. It assumes that fixed priority scheduling is being used and that arrival rates and execution times have little or no variability. RMA models are expressed in terms of sets of tasks, their overall topology and priorities, execution times, and the frequency with which external messages arrive. RMA theory includes formulae for computing worst-case latency and ensuring deadline satisfaction.

The use of an analytic theory to reason about a quality attribute is the underlying principle for a reasoning framework. The model representation is relative to the theory; the evaluation procedure is an application of the theory's logic to calculate specific quality attribute measures.

Note that inclusion of an analytic theory is more symbolic than literal. Reasoning framework documentation should identify and provide a short summary of the theory used, along with citations for more information, not reproduce the entirety of research in the area. Likewise, an implementation of a reasoning framework is likely to implement only as much of the theory as is used in the evaluation procedure, rather than all aspects of the theory.

Table 2 summarizes the analytic theories of the three reasoning frameworks introduced in Section 2.

Table 2: Analytic Theories of the Performance, Variability, and ComFoRT Reasoning Frameworks

Reasoning Framework	**Analytic Theory**
Performance	• queuing theory • scheduling theory
Variability	• impact analysis • cost estimation for instantiation mechanisms
ComFoRT	• finite state automata • temporal logic • model checking

2.3 Analytic Constraints

A reasoning framework is not intended to analyze any arbitrary design. By restricting the design space to which the reasoning framework is applied, more assumptions can be made in the analytic theory and evaluation procedure. These assumptions are often necessary to improve confidence in analytic results, permit the analytic tractability that allows the reasoning framework to be applied to larger systems, or even ensure solvability.

Analytic constraints are generated during the development of a reasoning framework as needed to achieve the desired analytic capability for a target class of systems. The Carnegie Mellon[®] Software Engineering Institute (SEI) has developed the co-refinement process as a means to manage the tradeoffs between the analytic constraints that limit the set of systems that can be analyzed and the generality of assemblies that can be created in the target platform [Hissam 02]. Briefly, the co-refinement process starts with as many analytic constraints as are necessary to thoroughly understand how to apply the analytic theory to a class of systems, even if the class is smaller than the class of interest. The process proceeds by incrementally relaxing analytic constraints and/or improving the analytic theory or evaluation procedure until the analytic constraints are sufficiently relaxed to allow the reasoning framework to be applied to an “interesting” class of systems of interest.

Each reasoning framework specifies the analytic constraints that must be met by the input architecture description. Only designs that meet the constraints are analyzable, as depicted conceptually in Figure 3. A simple example of an analytic constraint for a performance reasoning framework is “all event arrivals must be periodic.”

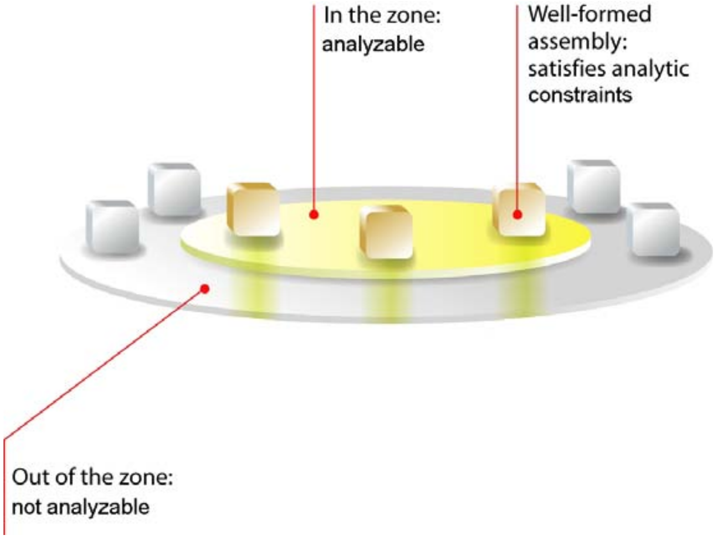

Figure 3: Analytic Constraints Define Analyzable Assemblies[4]

Additionally, a reasoning framework may also impose that some elements of the input design be associated with specific information needed to perform the analysis. For example, a performance reasoning framework can only be applied to architecture descriptions in which each component has an associated execution time. In some cases, this information may be obtained directly from architecture description—usually through annotations—but it can also be

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

[4] Analytically predictable systems lie within the set, or *zone*, of predictable assemblies. Assemblies that lie outside the zone are not analytically predictable.

derived from the response measure of quality attribute requirements that are used as inputs to the reasoning framework or even from the output of another reasoning framework.

Although we can express analytic constraints informally in prose, in the implementation of the reasoning framework they should be formally specified to enable automatic enforcement and verification.

Table 3 summarizes the analytic constraints of the three reasoning frameworks introduced in Section 2.

Table 3: Analytic Constraints of the Performance, Variability, and ComFoRT Reasoning Frameworks

Reasoning Framework	**Analytic Constraints**
Performance	• constrained interarrival rates for aperiodic streams • sporadic server aperiodic handler • components do not perform I/O operations and do not suspend themselves • components have fixed execution time
Variability	• applicable only to a specified set of product line instantiation mechanisms • cost of changing a component is a constant value and does not vary across any dimension
ComFoRT	• no shared data • static topologies • no pointer aliasing

2.4 Model Representation

The analytic theories used in reasoning frameworks are typically used in a broader context than just software architectures. As such, they have their own form of models,[5] which are abstractions of the system capturing only the information relevant to the analysis being performed. These models are represented within a reasoning framework using some encoding for elements and relations between elements, as well as properties. The encoding is suitable for machine processing.

The nature of the elements, relations, and properties is specific to each reasoning framework, as each uses a different view of the system. For example, a performance reasoning framework may deal with tasks that have execution time and deadlines as properties, while a reli-

[5] These models have been referred to as “quality attribute models” in other work. While these models are used to reason about quality attributes, they often are not specific to a quality attribute. For example, finite state automata can be used to reason about safety or reliability qualities.

ability reasoning framework may deal with processes that have mean time between failures (MTBF) as a property and dependencies among processes as relations.

Figure 4 shows a visualization of the model of some system in a performance reasoning framework. This example is based on RMA applied to a single processor on which multiple processes reside, each of which performs computations on its own input data stream. The various processes compete for the processor, with only one process being allocated the processor at a time. The quality attribute measure that can be calculated is latency—the time to completely process each message.

Models in the reasoning framework contain all the information that is required by the reasoning framework's evaluation procedure to calculate quality attribute measures. Because reasoning framework users are not expected to be experts in the underlying analytic theory or an accompanying notation, models are not supplied by users. Instead, models are generated automatically from architecture descriptions using the reasoning framework's interpretation. Though the primary purpose of a model representation is as an input to an automated evaluation procedure, a reasoning framework might also generate a visualization of the model that is more suitable for human consumption, such as that shown in Figure 4.

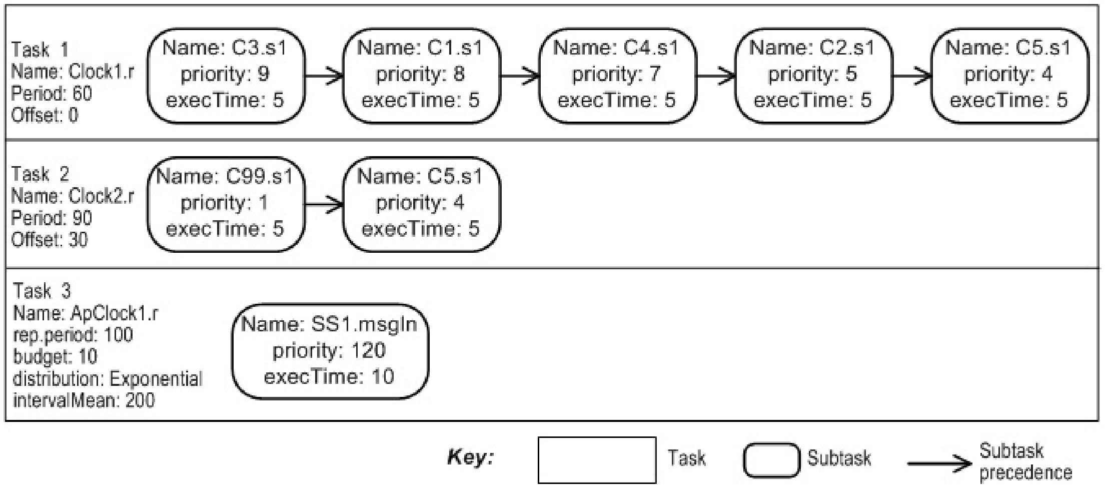

Figure 4: Graphical Representation of a Model Used in a Performance Reasoning Framework

Table 4 summarizes the model representations used in the three reasoning frameworks introduced in Section 2.

Table 4: Model Representations Used in the Performance, Variability, and ComFoRT Reasoning Frameworks

Reasoning Framework	**Model Representation**
Performance	queues, tasks, subtasks, and their relations
Variability	directed graph of dependencies annotated with costs and probabilities
ComFoRT	concurrent finite state machines represented in C and finite state processes (FSP)

2.5 Interpretation

Interpretation is the process used to generate a model representation suitable for consumption by a reasoning framework's evaluation procedure from an architecture description input to the reasoning framework. In an implemented reasoning framework, an interpretation is defined from the reasoning framework's input syntax (e.g., some architecture description language) to the input syntax of the implementation of the evaluation procedure.

Because the resulting model, not the architecture description, is the input to the evaluation procedure, it must contain all information that the evaluation procedure needs to calculate quality attribute measures. Consequently, the interpretation must be able to derive or extract this information from the supplied architecture description and any ancillary sources, such as concrete scenarios. One use of the reasoning framework's analytic constraints is to specify the information that must be provided for the reasoning framework to apply.

Using the previous example of a performance reasoning framework, the architectural description must include an identification of the schedulable entities, the execution times that contribute to each of them, and the period of each. The execution time of a process, for example, may be divided among various computations included within the process, and the interpretation must determine which computations are included in each process and the execution time of each computation.

The upper portion of Figure 5 shows an example of an architecture description expressed in CCL [Wallnau 03b] that is used as the input to a performance reasoning framework (λ_{SS}), and the lower portion shows the resulting performance model.

In this case, much of the architecture's structure (component boundaries, interfaces, and the like) has disappeared in the generation of a model of the underlying task structure. Despite the difference in appearance, the behaviors found in the reasoning framework model correspond to those found in the architecture, but with those aspects of the architecture that are not relevant to the analysis stripped away. Transformations of this degree are not unusual, but must be performed in a manner that ensures that the conclusions drawn from the model also apply to the architecture. The validation element of a reasoning framework helps ensure this correspondence. Validation is discussed in Section 3.1.2.

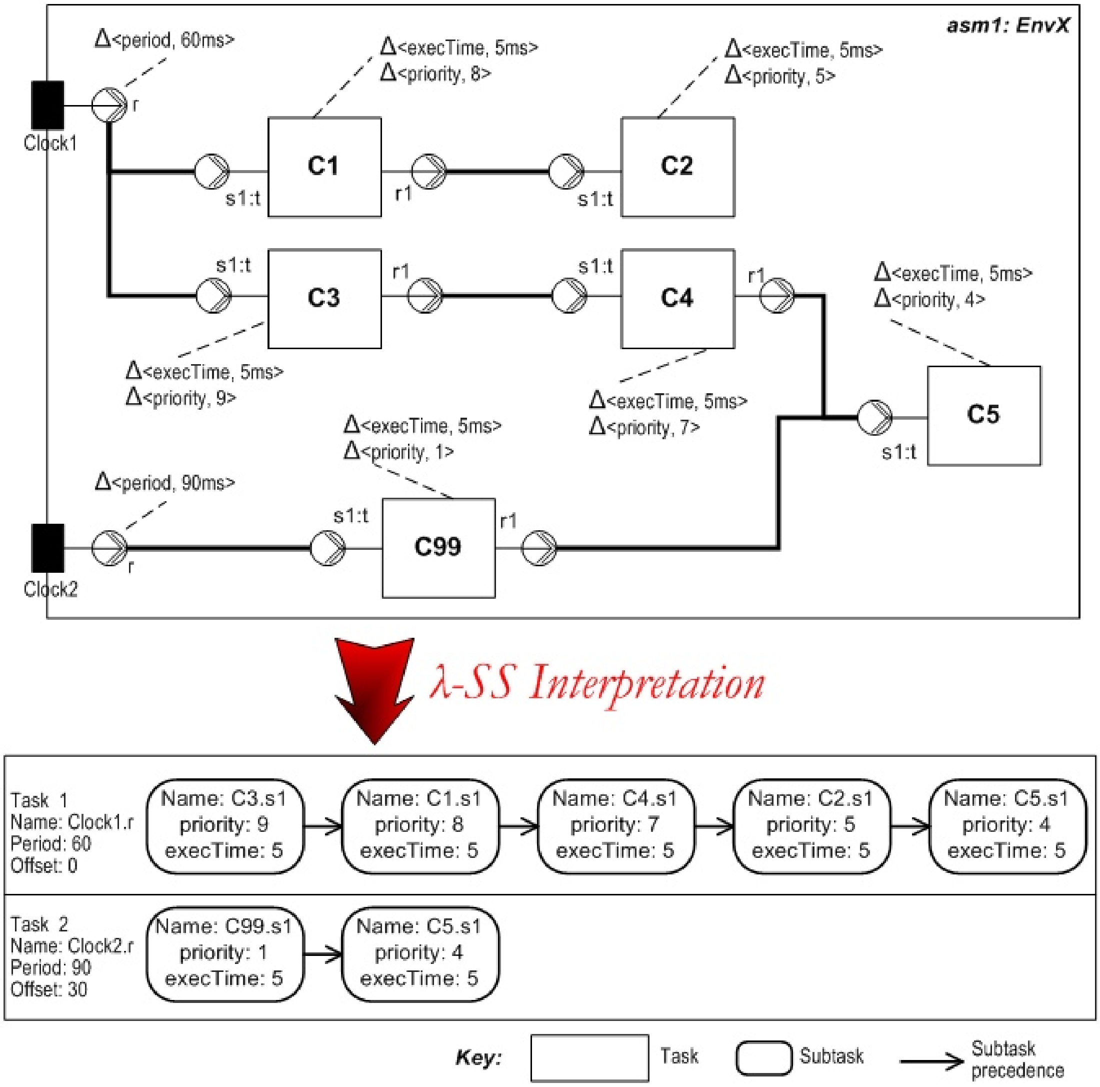

Figure 5: Model Generation for a Performance Reasoning Framework by Interpretation

Table 5 summarizes the interpretation of the three reasoning frameworks introduced in Section 2.

Table 5: Interpretations of the Performance, Variability, and ComFoRT Reasoning Frameworks

Reasoning Framework	**Interpretation**
Performance	from CCL[6] to queues, tasks, subtasks, and their relations
Variability	from an ArchE representation of • feature tree • mapping of feature tree to components • cost of change • probability of propagation between components to directed graph of dependencies annotated with costs and probabilities
ComFoRT	from CCL to concurrent finite state machines represented in C and FSP

2.6 Evaluation Procedure

An evaluation procedure[7] is the computable algorithm by which a reasoning framework calculates quality attribute measures from a model representation. An evaluation procedure may be implemented directly or indirectly, such as by the use of simulation models or spreadsheets.

A useful analogy is to think of the evaluation procedure as a deterministic procedure for calculating dependent parameters from independent ones. The dependent parameters are the quality attribute measures calculated using the reasoning framework. The independent parameters are those characteristics of the architecture that appear in the model representation; they are the knobs an architect can turn to affect how the system will behave with respect to the calculated quality attribute measures.

The following example is derived from RMA to show one form of evaluation procedure. In this example, RMA is applied to a single processor on which multiple processes reside, each of which performs computations on its own input data stream. The various processes compete for the processor, with only one process being allocated the processor at a time. The latency, the dependent parameter, is the time to completely process each message.

$$L_{n+1} = \sum_{j=1}^{i-1} \left\lceil \frac{L_n}{T_j} \right\rceil C_j + C_i + B_i$$

[6] In a future version of ComFoRT and the performance reasoning framework, additional languages (such as UML 2.0) will be supported.

[7] In other reports, this element has been referred to as a decision procedure, evaluation function, or model solver.

Three types of time are “experienced” by an arbitrary process under these circumstances: (1) preemption, (2) execution, and (3) blocking time. Preemption time is the contribution to latency attributable to higher priority processes. Blocking is the contribution to latency due to low-priority processes. Blocking time arises as a consequence of the shared resource topology.

In the model representation, C_i denotes the execution time of process i, T_i denotes the period of process i, and B_i denotes the blocking time incurred by process i. The evaluation procedure for calculating the worst-case latency for process i, assuming that processes 1 through i-1 are of higher priority, is an algorithm for iteratively solving the following formula until it converges (that is, the value of L_n remains the same for two consecutive iterations).

Table 6 summarizes the evaluation procedures of the three reasoning frameworks introduced in Section 2.

Table 6: Evaluation Procedures of the Performance, Variability, and ComFoRT Reasoning Frameworks

Reasoning Framework	Evaluation Procedure
Performance	• solving queuing formulae • simulation
Variability	solving cost equations
ComFoRT	model checking algorithms for verification combined with various state space reduction techniques

2.7 Summary of Element Examples

Table 7 summarizes how the six elements of a reasoning framework are realized in the performance, variability, and ComFoRT reasoning frameworks.

Table 7: Summary of the Elements of the Performance, Variability, and ComFoRT Reasoning Frameworks

	Performance	Variability	ComFoRT
Problem Description	prediction of average latency	prediction of the cost of creating a new variant	prediction of satisfaction of behavioral assertions expressed in SE-LTL
Analytic Theory	• queuing theory • scheduling theory	• impact analysis • cost estimation for instantiation mechanisms	• finite state automata • temporal logic • model checking

Table 7: Summary of the Elements of the Performance, Variability, and ComFoRT Reasoning Frameworks (continued)

	Performance	**Variability**	**ComFoRT**
Analytic Constraints	• constrained interarrival rates for aperiodic streams • sporadic server aperiodic handler • components do not perform I/O operations and do not suspend themselves • components have fixed execution time	• confined to a set of supported instantiation mechanisms • constant cost of change for components	• no shared data • static topologies • no pointer aliasing
Model Representation	queues, tasks, subtasks, and their relations	directed graph of dependencies annotated with costs and probabilities	concurrent finite state machines represented in C and FSP
Interpretation	from CCL to queues, tasks, subtasks, and their relations	from an ArchE representation of • feature tree • mapping of feature tree to components • cost of change • probability of propagation between components to directed graph of dependencies annotated with costs and probabilities	from CCL to concurrent finite state machines represented in C and FSP
Evaluation Procedure	• solving queuing formulae • simulation	solving cost equations	model checking algorithms for verification combined with various state space reduction techniques

3 Related Techniques

A reasoning framework encapsulates only the quality attribute knowledge needed to analyze some aspect of a system's behavior with respect to that quality attribute. It is method neutral—that is, it does not prescribe where inputs come from and what should be done with its outputs. This section illustrates a few techniques that can be used with reasoning frameworks, but by no means describes all possible reasoning framework uses.

3.1 Trusting Reasoning Framework Results

The most likely use of a reasoning framework is to guide development decisions, such as evaluating whether requirements are satisfiable, comparing design alternatives, and justifying reduced testing. To have any confidence that appropriate decisions are being made, however, engineers must know how much they can trust a reasoning framework's results. This depends on two factors: (1) the accuracy of the inputs to the reasoning framework and (2) the suitability and correctness of the reasoning framework's elements.

3.1.1 Certification

The degree to which the accuracy of the inputs to a reasoning framework can be established depends on how the reasoning framework is being used. When used as part of a design process, many inputs may be speculative or encompass a relatively large range of possible values. However, when some software elements are already implemented, inputs can potentially be very accurate. In such cases, a certification procedure can be used to place some objective measure of confidence on reasoning framework inputs.

Various technologies could be used for certification, depending on the type of information used as input to a reasoning framework. For example, if a performance reasoning framework requires the execution time of components as an input, this information can be gathered through observation over a selected sample of component executions. For a reasoning framework requiring detailed behavioral descriptions, a control flow graph extracted from source code can be compared to the behavioral model. In either case, some objective measure can be placed on the reasoning framework's inputs, which can be used to bound the confidence users should place in the reasoning framework's results.

3.1.2 Validation

In contrast, validation of a reasoning framework's outputs is less dependent on when a reasoning framework is applied. One important factor is the soundness of the reasoning framework's elements, which can and should be established during the creation of the reasoning framework. Additionally, reasoning framework results must be compared with the actual behavior of representative implemented systems to determine the level of confidence in the reasoning framework's results for a class of systems.

The major point of a reasoning framework is to establish a formal relationship between architectural designs and analytic theories so that the "reasoning power" of the theories can be used as the basis for (*a*) predicting behavior before the system is built, (*b*) understanding the behavior after it is built, and (*c*) making design decisions while it is being built and when it evolves. This formal correspondence between the architecture and its representation as a model in an analytic theory is achieved via interpretation. Therefore, one aspect of validation must be to ensure the correctness of the interpretation. Another aspect of validation is to ensure the correctness of the evaluation procedure.

Human inspection of the inference rules upon which the interpretation is based and proofs upon which the evaluation procedure is based is one way of validating the soundness of a theory. Simulation of the architecture to determine whether the analytic-based predictions are consistent with simulation results is another common technique.

Beyond the soundness of interpretation and evaluation, one must be concerned with the accuracy of predictions compared to how a fully fleshed out system executes on an actual platform. A reasoning framework that is platform independent, the norm in our work, will not contain any specific information about the accuracy of the analytic theories in the context of a specific platform. However, it should offer a procedure for validating the reasoning framework. One such procedure entails choosing a set of representative systems, executing those systems on a specific platform, and then comparing the actual results to the predicted results. These data provide the basis for generating statistically based levels of confidence in the predictions. The success of the validation procedure will depend on the care with which a sample set of systems is chosen.

3.2 Using a Reasoning Framework

There are two approaches to understanding and controlling quality attributes through the use of methods. One approach is to embed quality attribute knowledge into the method, and the other is to make the method, per se, independent of any particular quality attribute knowledge and to modularize the quality attribute knowledge so that any combination of quality attributes can be used with the method. An example of the first approach is the Rational Unified Process (RUP) [Kruchten 04], in which one step in architectural design is to decompose the system being designed into layers. Layers are a construct used to support portability and modifiability, and having an explicit step in the method that requires layers embeds portability and modifiability knowledge into the method.

Reasoning frameworks are an example of the second approach. Quality attribute knowledge is encapsulated into a reasoning framework, and all reasoning frameworks provide the same capability to the methods that rely on them. In our case, a reasoning framework contains the ingredients necessary to create predictive analytic models for understanding a specific quality attribute. The reasoning framework thus provides the capability to predict quality attribute behavior prior to implementing a system, which in turn allows the designer to try various design options before incurring implementation and refactoring expenses. This capability also allows a designer deeper insight into the ramifications of various design decisions. Since quality attributes have a significant impact on a system's architecture, the modeling capability provided by reasoning frameworks is a very important design aid.

Section 3.2.1 describes a way to use reasoning frameworks as part of an approach for recommending architectural changes to help meet quality attribute requirements. This approach is independent of any specific reasoning framework.

3.2.1 A Tactics-Based Approach

While the evaluation procedure of a reasoning framework gives a user calculated quality measures for his or her system, that is not enough to ensure that systems will meet their quality attribute requirements. Architects still have to decide what to do when calculated measures do not satisfy requirements.

The independent/dependent analogy for evaluation procedures provides insight into this decision process. Revisiting the performance example from Section 2.6, we see that there are exactly three independent parameters: (1) the period of a process, (2) the execution time of a process, and (3) the priority of a process. If a particular process does not meet its latency requirement, within the model there are only three things that can be manipulated to reduce the latency—these three parameters. Furthermore, given a fixed set of scheduling priorities, we know that reducing the execution time of a process or increasing the period of a process (not necessarily the process whose latency requirement is unmet) are the only options for reducing the latency of the process whose requirement is unmet.

Given a modification of an independent parameter (say reducing the execution time of a process), multiple architectural techniques can be used to effect this reduction. Any of the computations involved in the process, including algorithmic computation or overhead computations, can be reduced. Similarly, multiple architectural techniques have the effect of increasing the period of a process.

The possibility of changing independent parameters to affect the behavior of a system is the motivation for architectural tactics. An *architectural tactic* is a design decision that influences the control of a quality attribute response [Bass 03]. A tactic is used to transform a software architecture, where the result of the transformation changes a quality attribute measure to be closer to a desired value. An architectural tactic can change an architecture's structure (e.g., generalize modules) or one of its properties (e.g., reduce computational overhead), add new responsibilities that the architecture must support (e.g., introduce an intermediary), or add a new quality attribute requirement (e.g., introduction of a usability tactic will cause a performance requirement for feedback).

We have developed lists of architectural tactics for the attributes of availability, modifiability, performance, security, testability, and usability—partially through an examination of existing models and partially through interviewing experts in the various quality attributes. The parameters for the existing RMA reasoning framework are manipulated by some of the performance architectural tactics. We expect that as additional reasoning frameworks are developed for any of the attributes, the lists of architectural tactics will be refined and improved. The existing tactics for performance given in *Software Architecture in Practice* [Bass 03] are reproduced in Figure 6.

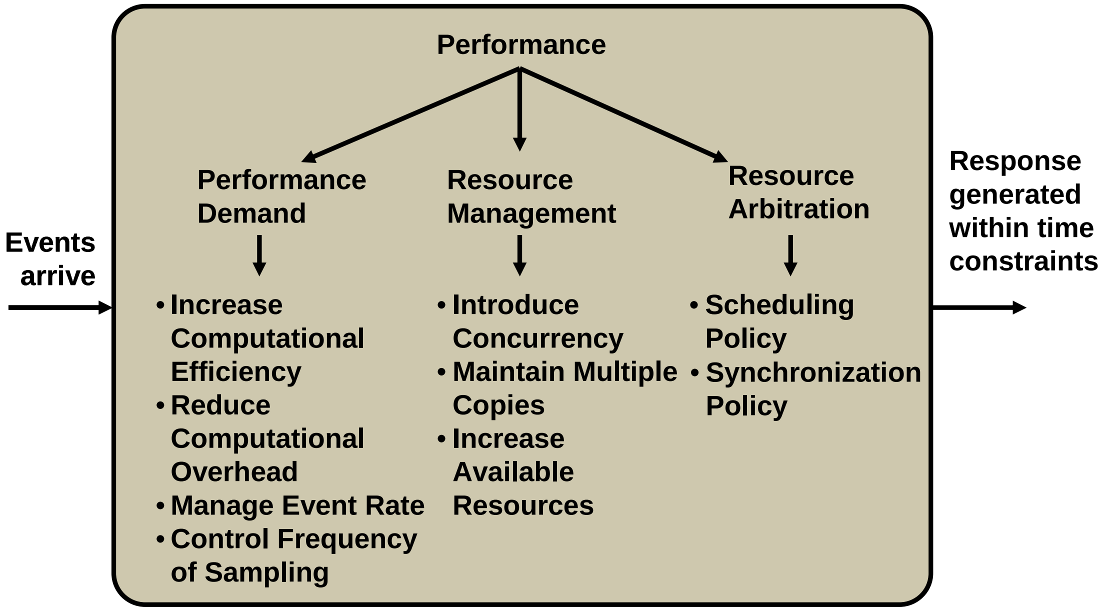
Performance
Performance Demand
Resource Management
Resource Arbitration
Events arrive
Response generated within time constraints
• Increase Computational Efficiency
• Reduce Computational Overhead
• Manage Event Rate
• Control Frequency of Sampling
• Introduce Concurrency
• Maintain Multiple Copies
• Increase Available Resources
• Scheduling Policy
• Synchronization Policy

Figure 6: Tactics for Performance

A tactics-based design manipulation approach matches a collection of tactics with rules for comparing calculated quality attribute measures to requirements and rules for selecting appropriate tactics to improve an architecture when requirements are not satisfied.

4 ComFoRT Description

ComFoRT is a reasoning framework used to determine whether systems satisfy behavioral assertions that can be used to characterize safety, reliability, or security requirements [Ivers 04]. These behavioral assertions, called claims, express properties in terms of patterns of events (communications) and state assignments (e.g., values of particular variables or modes of operation) and their relative ordering over time. Examples of claims include

- Industrial robots never enter a work area when sensors indicate a person is in the work area.
- Order acknowledgements are never lost or sent to the wrong receiver.
- Processes never write to the network after reading a file.

ComFoRT is based on model checking technology—a collection of algorithms used successfully, particularly in the hardware domain, to check whether a model of a system satisfies behavioral claims in *all possible executions*. The exhaustive nature of model checking provides higher confidence than is typically feasible to achieve using conventional testing techniques.

The ComFoRT reasoning framework has been developed by the SEI's Predictable Assembly from Certifiable Components (PACC) Initiative for inclusion in prediction-enabled component technologies (PECTs) [Wallnau 03a]. A PECT combines a component technology with one or more reasoning frameworks to provide a more predictable way to engineer systems from components. The component technology is used (among other things) to enforce the reasoning frameworks' analytic constraints, ensuring that any system that can be built using the PECT will have predictable behavior under its reasoning frameworks; that is, all systems built using the PECT will be predictable by construction with respect to a set of particular quality attributes or behavioral properties.

4.1 ComFoRT Elements

This section describes how the ComFoRT reasoning framework realizes the reasoning framework elements defined in Section 2.

4.1.1 Problem Description

Unlike other reasoning frameworks we are developing, ComFoRT is not tied to one specific quality attribute. It provides a general capability to verify behavioral assertions (claims) that can be used to reason about a variety of quality attributes, such as safety, reliability, and security. ComFoRT verifies whether the possible execution traces of a system satisfy claims expressed in SE-LTL [Chaki 04], which allows users to construct assertions from predicates over state values and events representing communication among components (for example, whenever component *C* receives an *E* event, *x* must be greater than 10). Therefore, ComFoRT solves problems that can be defined in terms of whether all system executions satisfy the types of claims expressible in SE-LTL, and the quality attribute measures are the truth or falsification (with evidence) of these claims.

4.1.2 Analytic Theory

ComFoRT is based on temporal logic model checking theory, which is in turn based on work in automata theory and temporal logics. Temporal logic model checking theory encompasses a number of techniques for efficiently determining whether a finite state automata satisfies claims written in a temporal logic under all possible executions. Whenever a model does not satisfy a claim, a counterexample is provided. A counterexample is a trace showing a specific sequence of steps undertaken by the processes in the model that lead to a state in which the claim is false.

The model checking theory used in ComFoRT deals only with the relative ordering of states and events among concurrent units, not with the passage of time.[8] During verification, every possible interleaving of concurrent units is considered when determining whether a claim is satisfied. Several different model checking techniques (such as abstraction, compositional reasoning, and symbolic representation) may be used to combat state space growth, allowing more complex models to be verified without creating resource problems, principally exhausting available memory and consuming excessive processing time.

4.1.3 Analytic Constraints

ComFoRT is designed for use with embedded, reactive systems. In addition to restricting applicability to components exhibiting reactive behavior, it imposes a number of analytic constraints that simplify verification, such as

- Concurrent units may not access shared data.
- Topologies must be static, including allocation of concurrent units.
- Component behavior may not be recursive, nor may cycles of unthreaded interactions exist in the system.
- The behavioral descriptions may not use pointer aliasing.

[8] Note that there are other forms of model checking, such as hybrid automata, that deal more explicitly with time, but at the cost of decreased tractability.

4.1.4 Model Representation

ComFoRT's model checking engine processes models represented using a combination of C and FSP files [Magee 01]. These languages are used together to provide a relatively concise model from which much larger finite state automata, expressed in more elementary primitives, are built. The C portion of these models captures the majority of the component behavior, including variables, computation, sequencing of actions, and links to the FSP portion of the model. The FSP portion of the model is used to describe the event-based communication among concurrent units.

4.1.5 Interpretation

ComFoRT includes an interpretation that generates model representations from architecture descriptions. Architecture descriptions that are used as inputs to ComFoRT are written in CCL (in the future, other languages will be available as alternatives to CCL). The generated model representations are written in a combination of C and FSP.

The interpretation extracts behavioral information needed to construct the automata represented in C and FSP from reaction descriptions, which in CCL are represented using a subset of UML statecharts. It extracts concurrency information from CCL keywords indicating where threads are present in the system and uses this information to allocate behavior descriptions to a collection of automata composed in parallel to reflect the concurrency the system will exhibit at runtime.

The interpretation ignores all information in the input CCL specification that is not related to constructing these automata, such as annotations of execution time or thread priority.

4.1.6 Evaluation Procedure

ComFoRT's evaluation procedure is a model checking engine that implements a collection of model checking algorithms, including state space reduction techniques such as automated forms of predicate abstraction and assume/guarantee reasoning. The algorithms are particularly appropriate for the class of systems we target.

A critical part of ComFoRT's evaluation procedure is its use of automated predicate abstraction to minimize state space growth. With the assistance of theorem provers, it computes conservative abstractions of the concrete model (the model representation). The nature of the conservative abstractions ensures that if an abstraction, which can be verified much more efficiently, satisfies a claim, then the concrete model must also satisfy it.

This approach is combined with a counterexample-guided abstraction refinement (CEGAR) approach to control the generation of abstract models. Should verification find that an error is spurious (a result of the abstraction, not necessarily an error of the concrete model), the counterexample indicating the error is used to refine the abstraction, and verification is performed over the new abstract model. The result is an approach that begins with a small abstraction that is incrementally enlarged as needed until the claim is verified or refuted.

4.2 ComFoRT Implementation

Figure 7 shows how the elements of ComFoRT are realized in an implementation. The tool implementing interpretation generates C and FSP files (the model representation) based on a CCL specification (the architecture description). This tool also implements checks to statically determine whether a given CCL specification satisfies all analytic constraints. The model checker, Copper, implements the evaluation procedure, determining whether a particular claim (desired quality attribute measure) holds for the provided model representation.

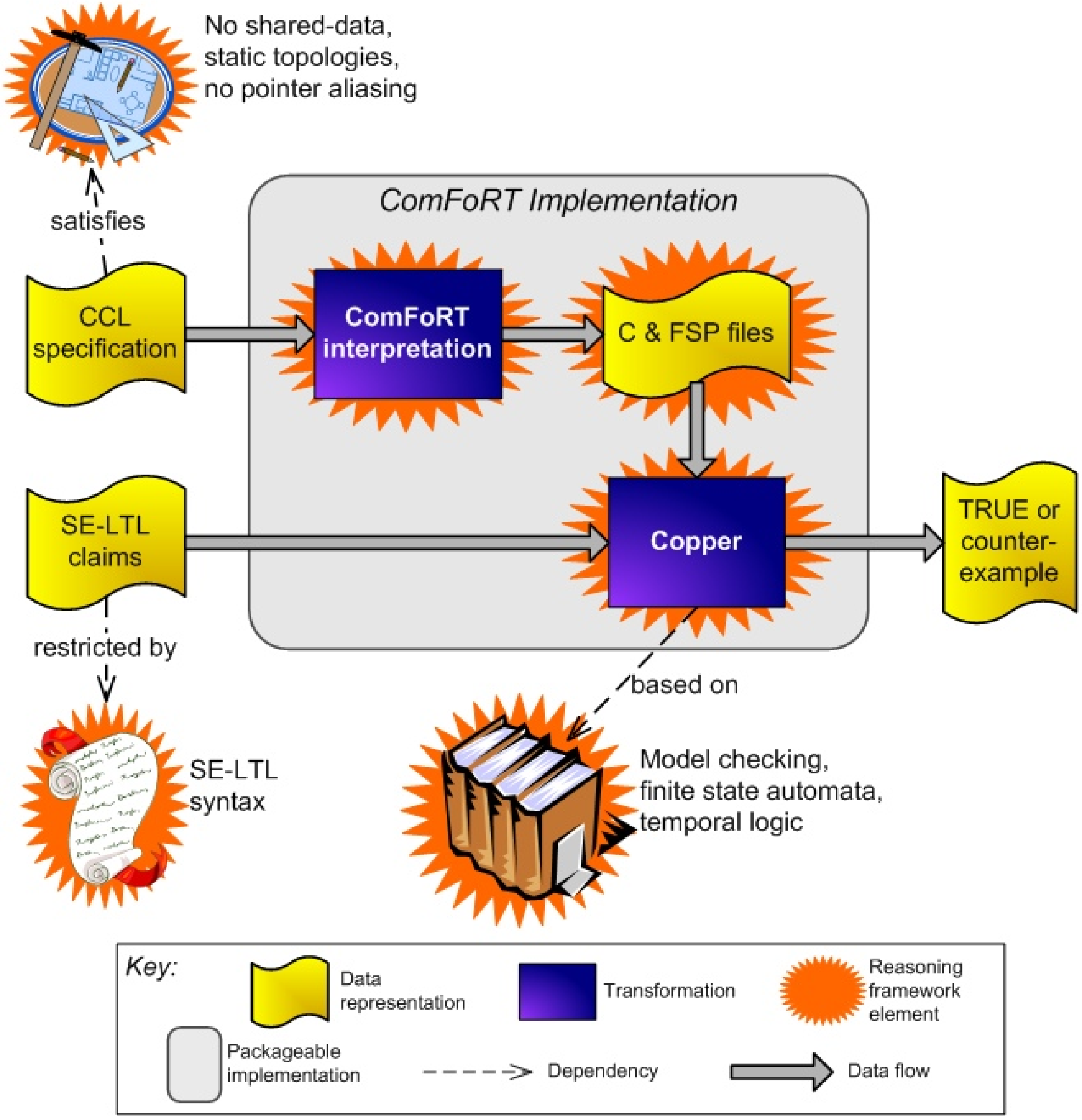

Figure 7: ComFoRT Implementation

5 Summary

Determining whether a system will satisfy critical quality attribute requirements in areas such as performance, modifiability, and reliability is a complicated task that often requires the use of many complex theories and tools to arrive at reliable answers. This report described a vehicle for encapsulating the quality attribute knowledge needed to understand a system's quality behavior as a reasoning framework that can be used by nonexperts. A reasoning framework establishes a formal relationship between architectural designs and analytic theories so that the "reasoning power" of the theories can be used as the basis for (*a*) predicting behavior before the system is built, (*b*) understanding the behavior after it is built, and (*c*) making design decisions while it is being built and when it evolves. This report defined the elements of a reasoning framework and illustrated the concept by describing several specific reasoning frameworks and how they realize these elements.

References

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

[Bachmann 03] Bachmann, F.; Bass, L.; & Klein, M. *Preliminary Design of ArchE: A Software Architecture Design Assistant* (CMU/SEI-2003-TR-021, ADA421618). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports/03tr021.html.

[Bass 03] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice, Second Edition*. Boston, MA: Addison-Wesley, 2003.

[Chaki 04] Chaki, S.; Clarke, E.; Ouaknine, J.; Sharygina, N.; & Sinha, N. “State/Event-Based Software Model Checking,” 128-147. *Integrated Formal Methods 4th International Conference (IFM 2004)* (in Lecture Notes in Computer Science [LNCS], Volume 2999). Canterbury, Kent, UK, April 4-7, 2004. Berlin, Germany: Springer-Verlag, 2004.

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

[Ivers 04] Ivers, J. & Sharygina, N. *Overview of ComFoRT: A Model Checking Reasoning Framework* (CMU/SEI-2004-TN-018). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents/04.reports/04tn018.html.

[Kruchten 04] Kruchten, P. *The Rational Unified Process: An Introduction, Third Edition*. Boston, MA: Addison-Wesley, 2004.

[Magee 01] Magee, J. & Kramer, J. *Concurrency: State Models & Java Programs*. West Sussex, England: Wiley Publications, 2001.

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

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

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY
(Leave Blank)

2. REPORT DATE
July 2005

3. REPORT TYPE AND DATES COVERED
Final

4. TITLE AND SUBTITLE
Reasoning Frameworks

5. FUNDING NUMBERS
F19628-00-C-0003

6. AUTHOR(S)
Len Bass, James Ivers, Mark Klein, Paulo Merson

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

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-2005-TR-007

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-2005-007

11. SUPPLEMENTARY NOTES

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

12B DISTRIBUTION CODE
41

13. ABSTRACT (MAXIMUM 200 WORDS)

Determining whether a system will satisfy critical quality attribute requirements in areas such as performance, modifiability, and reliability is a complicated task that often requires the use of many complex theories and tools to arrive at reliable answers. This report describes a vehicle for encapsulating the quality attribute knowledge needed to understand a system's quality behavior as a reasoning framework that can be used by nonexperts. A reasoning framework includes the mechanisms needed to use sound analytic theories to analyze the behavior of a system with respect to some quality attribute. This report defines the elements of a reasoning framework and illustrates the reasoning framework concept by describing several reasoning frameworks and how they realize these elements.

14. SUBJECT TERMS
reasoning framework, quality attribute behavior, software architecture

15. NUMBER OF PAGES
40

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