Integrating Quality-Attribute Reasoning Frameworks in the ArchE Design Assistant

Andres Diaz-Pace, Hyunwoo Kim, Len Bass, Phil Bianco, and Felix Bachmann

Software Engineering Institute, Carnegie Mellon University
4500 Fifth Avenue, Pittsburgh, PA-15213-2612, USA
{adiaz,hkim,ljb,pbianco,fb}@sei.cmu.edu

Abstract. Techniques and tools for specific quality-attribute issues are becoming a mainstream in architecture design. This approach is practical for evaluating the architecture in early stages but also for planning improvements for it. Thus, we believe that one challenge is the integration of the individual capabilities of quality-attribute techniques. This paper presents our research work on a design assistant called *ArchE* that, based on reasoning framework technology, provides an infrastructure for third-party researchers to integrate their own quality-attribute models. This infrastructure aims at facilitating the experimentation and sharing of quality-attribute knowledge in both research and educational contexts.

Keywords: Architecture-based analysis & design, quality attributes, design assistance, *ArchE*.

1 Introduction

The importance of tackling quality-attribute requirements (e.g., performance, modifiability, reliability and other "non-functional" issues) in early development stages has been widely recognized by the software community. The software architecture is an effective instrument to reason about the relationships between design decisions and quality attributes [4].

One mechanism for modeling quality-attribute issues is via reasoning frameworks. A *reasoning framework* [5] is an abstraction to encapsulate the knowledge needed to understand and estimate the behavior of a system with respect to a particular quality, so that this knowledge can be applied by non-experts. Having encapsulated models for quality attributes has advantages in terms of scale and level of detail, because it helps the architect to manage the relationships among multiple quality-attribute models when designing an architecture. Ideas of the same kind have been discussed by other researchers as well [7, 10, 14].

In this context, automated tool support is crucial to take advantage of quality-attribute knowledge. A particular category of tools is design assistants. A *design assistant* can be seen as an agent that supports the architect in decision-making, either by making suggestions on possible courses of action or by performing some computations autonomously on her behalf. For several years, the Software Engineering Institute

(SEI) has been developing an assistant for architecture design called *ArchE*[1] [1, 2, 16]. In a nutshell, this prototype performs a semi-automated search of the design space, using the outputs of reasoning frameworks to direct the search towards solutions with known quality properties. The initial release of *ArchE* consisted of a rule-based engine and examples of reasoning frameworks that allow the user to explore simple architectures for performance and modifiability.

However, the challenge is not only about sound reasoning frameworks able to link architectures to quality-attribute models individually. In order to fully realize the potential of this technology, we argue that a design assistant should allow people to put their own reasoning frameworks to work together. In this paper we describe an extension of *ArchE* called *ArchE Reasoning Framework Interface (ArchE-RF Interface)* to support such an objective. This new release consists of a collaborative infrastructure for third parties to contribute reasoning frameworks to *ArchE* as plugin modules. The approach is based on a blackboard organizational style, in which the *ArchE* engine plays the role of control component and the reasoning frameworks register themselves with *ArchE* through a publish-subscribe schema. *ArchE* has no semantic knowledge of quality-attribute models; it just manages the basic inputs such as scenarios and responsibilities, delegates the design work to the available reasoning frameworks, and then assembles their results.

The contribution of this approach is that a researcher can concentrate directly on the modeling and implementation of a reasoning framework for her quality of interest, and afterwards instantiate her reasoning framework easily on top of the *ArchE-RF Interface*. Furthermore, providing a platform for modular reasoning frameworks that are *ArchE*-compatible, we expect to support the development and integrated use of quality-attribute models by researchers, practitioners and educators.

The rest of the paper is structured around 5 sections. Section 2 describes the key concepts of the *ArchE* vocabulary for reasoning frameworks. Section 3 is devoted to the interactions between *ArchE* and the reasoning framework plugins using the *ArchE-RF Interface*. Section 4 briefly describes our experiences implementing two reasoning framework examples. Section 5 comments on related work. Finally, Section 6 presents the conclusions and discusses future lines of work.

2 Reasoning Frameworks: The Building Blocks

Conceptually, a reasoning framework is a modular entity that provides the capability to reason about specific quality-attribute behavior(s) of an architecture. In its original formulation [5], a reasoning framework only involved analytic theories (e.g., queuing networks for performance, change impact for modifiability, Markov chains for availability, etc.) to determine whether an architecture satisfies quality-attribute requirements. Later, this formulation was extended with the capability to transform an architecture using tactics [2] in order to satisfy unmet quality-attribute requirements.

The class of behaviors or situations for which the reasoning framework is useful is referred to as the problem description. A specification of a problem description can be a collection of scenarios along with an initial architectural model for the system. The

[1] http://www.sei.cmu.edu/architecture/arche.html

analytic theory needs also a representation to abstractly describe those aspects of the design we should reason about. This representation is referred to as the analysis model. In this context, a reasoning framework is expected to support three phases [2]:

1. **Interpretation:** The mapping procedure that converts the architectural model into the analysis model
2. **Evaluation:** The procedure used to solve the analysis model and compute quality-attribute measures for the scenarios. These measures help to determine whether the current architecture satisfies its scenarios.
3. **Re-design (optional):** In case some scenarios are unmet, tactics permit to adjust the structure/behavior of the current architectural model.

To accomplish these phases, the process of building a reasoning framework relies on a vocabulary of architectural concepts. The key concepts we have used for the development of *ArchE-RF Interface* include: general quality-attribute scenarios, concrete quality-attribute scenarios, quality-attribute models, responsibilities, architectural tactics, and architectural views. Figure 1 shows the ontology of concepts and the relationships among them.

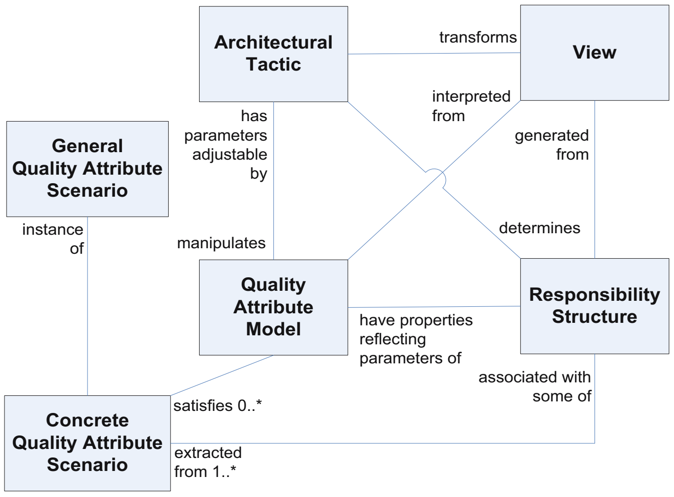


Fig. 1. Ontology of architectural concepts for reasoning frameworks

A summary of the concepts is provided below (see references for further details).

- *General quality-attribute scenario.* A system-independent table for deriving quality-attribute requirements. The table consists of six parts, namely: a stimulus, a stimulus source, an environment, an artifact being stimulated, a response, and a response measure; each part having different possible values. General scenarios for several quality attributes are discussed in [4].
- *Concrete quality-attribute scenario.* A system-specific requirement that is an instance of a general scenario. A concrete scenario for modifiability would look like "The operating system used by different customers may vary (stimulus).

Adaptation of the software to the different processors (response) should be done within 1 person-day (response measure)".

- *Quality-attribute model.* The result of interpreting an architecture design with an analytic theory. A quality-attribute model usually has a set of independent parameters that can be manipulated (in specific reasoning framework instances) to control the values of the measures produced by the evaluation procedure. See the chain impact analysis theory described in Section 2.1 for an example.
- *Responsibilities.* A responsibility is an activity undertaken by the software being designed [18]. We use responsibilities as a means to express functional requirements as a part of quality-attribute scenarios, and moreover, as a means to integrate the models produced by various reasoning frameworks. Responsibilities can be annotated with quality-specific properties or take part in relationships. All this information provides clues for a reasoning framework to create an initial architecture and reason about quality-attribute issues. See example of Section 2.1.
- *Architectural tactic.* A vehicle for satisfying a quality-attribute-response measure by manipulating some aspect of a quality-attribute model through design decisions. That is, a tactic is an architectural transformation based on a quality-attribute justification. A tactic comes with both analysis rules and design rules. The former rules specify how the independent parameters of a quality-attribute model can be controlled to achieve a desired measure (i.e., a scenario response). The latter rules codify architectural decisions to move from a given architecture to another one (variant) with a better fitness. See example of Section 2.1.
- *Architectural view.* A view is a design structure of the system that can be seen from a viewpoint [4]. In general, an architectural view can be seen as a typed graph that is composed of architectural design elements, their properties, and their relations for the viewpoint. Examples of common architectural views are: the module view, the process view, the component-and-connector view, etc.

Note that the ontology involves three types of model transformations. A first type of transformation generates the architectural model (i.e., a set of architectural views) from the scenarios and responsibilities. Then, a second type of transformation is the interpretation procedure, which translates the views to a representation that is more suitable for quality-attribute analysis. Finally, a third type of transformation is that of tactics, which modifies the current architectural view(s) to generate architectural variants. Here, it is assumed also that the tactics determine responsibilities and relationships for the architecture, which are consistent with the quality-attribute models manipulated in terms of its parameters.

In addition, we require every reasoning framework to publish a *manifesto*. This manifesto is used by *ArchE* to integrate the reasoning framework to the infrastructure, checking compliance of its modeling concepts and detecting possible conflicts with other reasoning frameworks. The manifesto specifies the quality attribute the reasoning framework is interested in, the scenario structure, and other architectural element types that the reasoning framework is able to process.

2.1 Example: A Modifiability Reasoning Framework

We briefly describe a modifiability reasoning framework based on change impact analysis (CIA) [6], as an example of the kind of quality-attribute models that can be integrated in *ArchE* [1][2]. Modifiability is seen as "the ability of a software architecture to accommodate changes". Given a set of change scenarios, the level of modifiability of an architecture is a function of how functionality is allocated to modules and how these modules interact with each other.

According to the CIA theory, the architecture is interpreted as a graph, in which the nodes correspond to "units of change" (e.g., responsibilities, modules, interfaces) while the arcs represent dependencies between the nodes (e.g., functional dependencies, data flows). A modification of a specific node is likely to propagate to a neighborhood of nodes. We assume that the effects of the change in the neighbors decrease as a function of the distance to the source of the change. So, we define an evaluation procedure that traverses the graph and returns cost estimations for the change. To do this evaluation, nodes and arcs are annotated with properties. The total cost of making a change is computed as a weighted sum that considers the costs of individual nodes and the probabilities of change rippling associated to the arcs. Furthermore, we allow manipulation of the graph via tactics, so as to affect the results of the evaluation function. This is accomplished either by adjusting the values of properties or by altering the topology of the graph.

Figure 2 outlines the manifesto for our modifiability reasoning framework. This manifesto is an XML file that lists the element types handled by the reasoning framework. The manifesto exposes structural information of the element types, but it is not concerned with their behavior. The first part of the manifesto identifies the reasoning framework itself (tag <*rf*>). For CIA, the manifesto specifies a new type of modifiability scenario (section <*scenarioTypes*>) as well as modifiability-related elements for it (e.g., sections for responsibility parameters, responsibility relationship types, view element types, view relation types, etc.). *ArchE* will use this specification as "meta-information" of what is needed by the reasoning framework to operate. Additionally, *ArchE* will display appropriate GUIs and infer the data mappings to its database.

In the <*responsibilityStructure*> section, we specify that a responsibility can take part in a "functional dependency" relationship with other responsibilities. Besides, we decorate plain responsibilities and dependency relationships with modifiability-specific parameters. One parameter of a responsibility is the cost of changing that responsibility. Two parameters of a dependency are the probabilities for "incoming" and "outgoing" rippling of changes. The assignment of values to these parameters is done by the architect based on previous experiences or empirical data.

The <*view*> section specifies a module view [4] as a suitable architectural description for modifiability issues. A module can be thought of as a code or implementation unit that delivers some functionality. Modules have relationships with other modules. A common relationship between modules is dependency, which denotes coupling between two modules. Since *ArchE* relies on responsibilities, we have extended the module view to include allocation relationships, so that a module can support one or more responsibilities. Dependencies between modules are computed in terms of

[2] Although the CIA-based model is plausible to reason about modifiability, the model has not been fully validated yet.

responsibility dependencies and responsibility allocations. That is, if a responsibility A is dependent on a responsibility B and they are allocated to different modules MA and MB respectively, we will have then a dependency between modules MA and MB. The dependency relationship for modules behaves similarly to the responsibility dependency, having associated probabilities for incoming and outgoing change rippling. The <*model*> section is about the representation of the graph in terms of units of change and rippling probabilities. This section is optional in the manifesto, and it only serves to visualization purposes of the *ArchE* GUI.

```
<!--xml header -->
<rf   <!-- Reasoning framework identification -->
      id="ChangeImpactModifiabilityRF"            <!-- Unique ID -->
      quality="Modifiability"                     <!--Target quality attribute -->
      name="ModifChangeImpact  RF v0.1"           <!--Description -->
      version="0.1"                               <!-- Version of this reasoning framework -->
>
 <scenarioTypes> <!-- Specification of 6-part general scenario -->
 . . .
 </scenarioTypes>
 <responsibilityStructure > <!-- Information about responsibility parameters,  types of responsibility
relations  and parameters for those relationships,  e.g., dependency relationship, cost of change or
rippling properties -->
    <parameterTypes> . . . </parameterTypes>
    <responsibilityParameters> . . . </ responsibilityParameters >
    <relationshipTypes> . . . </ relationshipTypes >
 </responsibilityStructure >
 <view > <!-- Description of the design elements and relationships used in the architectural
representation, e.g., a module view-->
    <viewElementType> . . . </ viewElementType >
    <viewRelationType> . . . </ viewRelationType >
 . . .
 </view >
 <model > <!-- Description of elements and relationships of the model used  for quality- attribute
analysis, e.g., a dependency graph -->
    <modelElementType> . . . </ modelElementType >
    <modelRelationType> . . . </ modelRelationType >
 . . .
 </model >
</rf>
```

Fig. 2. Fragment of the XML manifesto

When the reasoning framework executes, its interpretation procedure will filter out those design elements and design relations of the module view that are related to scenario-specific responsibilities, in order to construct a graph for the architecture. This graph will be evaluated according to a cost formula. We used a cost formula derived from [1] for computing the cost of all the nodes impacted by a given scenario. The interpretation and evaluation are graphically exemplified in Figure 3. Finally, the design cycle is completed with two modifiability tactics [3], which are not included in the manifesto but supported by the reasoning framework implementation. The first tactic aims at reducing the cost of modifying a single (costly) responsibility by splitting it into children responsibilities. An instance of this tactic is shown at the bottom of Figure 3. The second tactic aims at reducing the coupling between modules by inserting an intermediary that breaks module dependencies. These tactics are materialized through transformations that affect both the module view and the responsibility structure. The re-interpretation of the architectures generated by the transformations leads to slightly different dependency graphs, and consequently, the modifiability measures for these graphs vary. The process of *interpretation-evaluation-transformation* continues until the analysis of the scenarios reaches values that satisfy the architect's expectations.

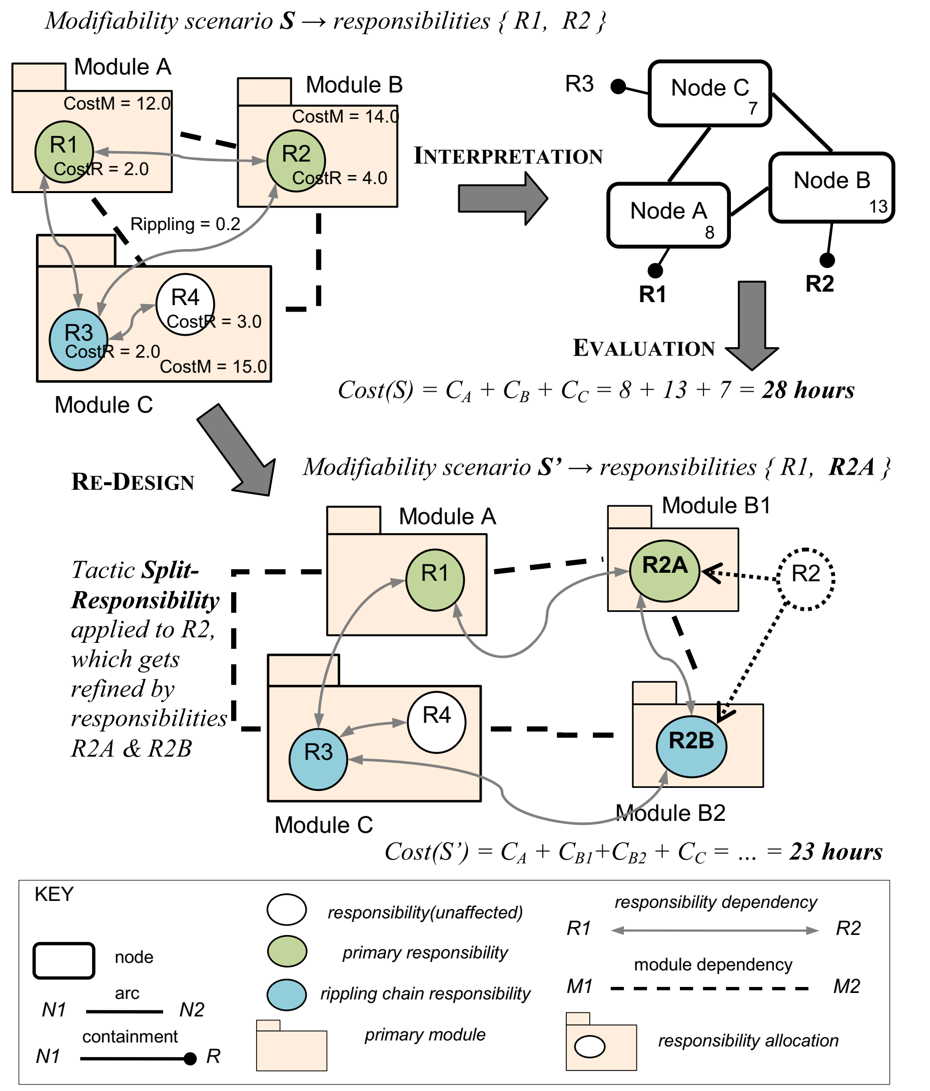

Modifiability scenario S → responsibilities { R1, R2 }
Module A
CostM = 12.0
R1
CostR = 2.0
Module B
CostM = 14.0
R2
CostR = 4.0
Rippling = 0.2
R4
CostR = 3.0
R3
CostR = 2.0
CostM = 15.0
Module C
INTERPRETATION
R3
Node C
7
Node A
8
Node B
13
R1
R2
EVALUATION
Cost(S) = C_A + C_B + C_C = 8 + 13 + 7 = 28 hours
RE-DESIGN
Modifiability scenario S' → responsibilities { R1, R2A }
Tactic Split-Responsibility applied to R2, which gets refined by responsibilities R2A & R2B
Module A
R1
Module B1
R2A
R2
R4
R3
R2B
Module C
Module B2
Cost(S') = C_A + C_B1 + C_B2 + C_C = ... = 23 hours
KEY
node
arc
N1
N2
containment
N1
R
responsibility(unaffected)
primary responsibility
rippling chain responsibility
primary module
responsibility dependency
R1
R2
module dependency
M1
M2
responsibility allocation

Fig. 3. Interpretation, evaluation and re-design for modifiability

3 ArchE-RF Interface: The Collaborative Infrastructure

The working of *ArchE* follows a blackboard style [8], in which different actors collaborate to produce a solution for a problem. Each actor can potentially read information from the blackboard that was developed by other actors; and conversely, each actor can introduce new information into the blackboard that could be of interest to anyone else. The reasoning frameworks can be seen as knowledge sources, and *ArchE* is the control component that manages the interactions among them, so as to ensure progress in the architecting process. Note that *ArchE* is an assistant to explore quality-driven architectural solutions, rather than being an automated design tool. Since not all the decisions can be made by *ArchE*, the user becomes an additional actor in the schema, who makes the final decisions. For instance, the computations of the reasoning frameworks need human intervention for specifying correct scenarios, entering the necessary parameters for analysis and tactics, among other tasks. This modality of assistance is known as *mixed-initiative* [17].

Enhancing the assistive capabilities of *ArchE* means to integrate different reasoning frameworks into the blackboard schema. To do so, we have re-designed the initial version of *ArchE* towards a collaborative infrastructure: the *ArchE Reasoning Framework Interface (ArchE-RF Interface)*. In this infrastructure, reasoning frameworks are considered as “external plugins”. The term “external” means that a reasoning framework resides anywhere outside the *ArchE* process, even on a remote machine over networks. The term “plugin” means that a reasoning framework can be added or removed at runtime without disturbing the current tasks of *ArchE*. Thus, *ArchE* can take advantage of multiple computing resources by executing reasoning frameworks in parallel.

In Figure 4, we show a simplified view of the interactions between *ArchE* and the reasoning frameworks. A reasoning framework announces itself in the infrastructure via its manifesto, and *ArchE* enables the reasoning framework for operation. From that point on, the *ArchE* engine starts sending asynchronous interaction commands to the reasoning framework(s), and also communicating information through a database. Meanwhile, each reasoning framework acts as a “command listener”, executing the received commands with its own logic and accessing the database. Once a reasoning framework has successfully executed a command, it sends the results back to *ArchE*. Examples of command results can be: analysis values, suggested tactics, or questions for the user. *ArchE* either waits for the results of a predefined command or proceeds with other commands, depending on the context.

The collaborative infrastructure relies on four main components:

- *ArchE Engine*. This component retains the functionality of the first release with respect to the general structure of the search for architectural alternatives. The only modification is that the design work is now delegated to “remote” reasoning frameworks. This engine has very little knowledge of either quality-attribute design techniques or semantics of the system being designed. The responsibilities of the engine are: processing of user inputs, update of GUI panels, parsing of the manifesto, coordination of reasoning frameworks, presentation of their results, and display of user questions.

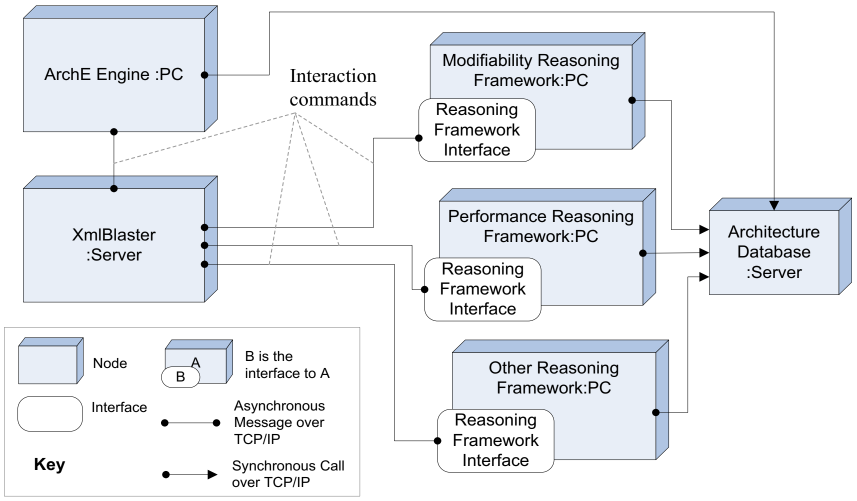

ArchE Engine :PC
Interaction commands
Modifiability Reasoning Framework:PC
Reasoning Framework Interface
XmlBlaster :Server
Performance Reasoning Framework:PC
Reasoning Framework Interface
Architecture Database :Server
Other Reasoning Framework:PC
Reasoning Framework Interface
Node
A
B
B is the interface to A
Interface
Asynchronous Message over TCP/IP
Key
Synchronous Call over TCP/IP

Fig. 4. Integration of external reasoning frameworks with the ArchE engine

- *XmlBlaster*[3]. This is a message-oriented middleware where implicit message invocations can take place among participants over networks. This middleware fosters extensibility in terms of adding (or removing) a participant without considering others.
- *Reasoning Framework Interface*. This is the actual interface to a reasoning framework. It abstracts the details about working with *XmlBlaster*, the communication protocol between *ArchE* and the reasoning framework, and also the algorithms executing the interaction commands.
- *Architecture Database*. This repository is used to manage any persistent data that need to be shared by *ArchE* and all participating reasoning frameworks. The data include both the original and the candidate architectures (e.g., scenarios, responsibilities, architectural views, and relationships among them).

The *ArchE-RF Interface* is implemented in Java, so the reasoning framework functionality must be implemented in Java as well. Anyway, given the *XMLBlaster* characteristics, the functionality could be implemented in other programming language (e.g., C or C++) and then assembled with the top-level Java code using JNI[4].

3.1 ArchE Interaction Commands

Basically, *ArchE* runs a search algorithm for finding promising candidate architectures. The search is divided between the *ArchE* engine and the available reasoning frameworks. On one side, the engine controls the main search cycle and makes a global evaluation of the proposals of the reasoning frameworks. On the other side, each reasoning framework should implement its own search algorithms to suggest tactics for the current architecture.

[3] http://www.xmlblaster.org/
[4] http://java.sun.com/javase/6/docs/technotes/guides/jni/

The search cycle is structured around five commands that govern the interactions with the reasoning frameworks.

- *ApplyTactics.* This command requests a specific reasoning framework to apply a tactic to the current architecture in order to refine it (*Re-design* phase). The tactic must come from a question that was previously shown to the user of *ArchE* and she agreed to apply (see command *DescribeTactic* below). The expected result is to have the refined version of the current architecture in the database.
- *AnalyzeAndSuggest.* This command requests a reasoning framework to analyze the current architecture regarding scenarios of interest, and to suggest new tactics if some scenarios are not fulfilled (*Interpretation* and *Evaluation* phases). The reasoning framework returns the analysis results and the tactics (if any) to *ArchE.*
- *ApplySuggestedTactic*. This command requests a reasoning framework to apply a tactic to the current architecture in order to create a new candidate architecture (*Re-design* phase on a new architecture instance). The tactic must be one of the tactics that the reasoning framework suggested when executing the *AnalyzeAnd-Suggest* command. The expected result is to have a candidate architecture in the database.
- *Analyze*. This command requests a reasoning framework to analyze a candidate architecture regarding scenarios of interest (*Interpretation* and *Evaluation* phases on a new architecture instance). The evaluation results returned by the reasoning framework will be used by *ArchE* to prioritize candidate architectures.
- *DescribeTactic*. This command requests a reasoning framework to provide *ArchE* with user-friendly questions that describe tactics or any other recommendations. This is actually the main mechanism to offer design advice to the user on how to improve its architecture. Again, *ArchE* does not know about the semantics of user questions, it just shows these questions in the GUI and let the user decide.

Whenever the user makes a change to some part of the design, *ArchE* starts a new cycle of its algorithm and executes the above commands in the following sequence:

1. If the change is a decision to apply a tactic, *ArchE* sends *ApplyTactics* to the reasoning framework that suggested the tactic, and then, the reasoning framework modifies the working architecture according to the tactic. For example, let's consider that our modifiability framework inserts an intermediary module upon user's request.
2. For every reasoning framework, *ArchE* sends *AnalyzeAndSuggest* sequentially. Each reasoning framework might modify the current architecture (if needed), in preparation for the following analysis task. This assures consistency on the responsibility structure and initialization of its architectural view. For example, our reasoning framework can decorate new responsibilities with costs (if that property is missing) and update the module view by allocating every new responsibility to a module. Then, each reasoning framework starts its analysis of the architecture. If the analysis results say that some scenarios are not fulfilled, it tries to find tactics suitable for the architecture. At last, it returns the analysis results and the list of suggested tactics. For instance, our reasoning framework may run its change impact analysis, detect a costly responsibility as a main contributor to the scenario response (total cost), and propose a responsibility splitting.

3. For every suggested tactic:
   a) *ArchE* sends *ApplySuggestedTactic* to the reasoning framework with the tactic under consideration. The reasoning framework creates a candidate architecture by modifying the architecture according to the tactic.
   b) For every reasoning framework, *ArchE* sends *Analyze* in parallel. Each framework analyzes the candidate architecture and returns the evaluation results to *ArchE*.
4. *ArchE* prioritizes all the evaluation results that came from applying suggested tactics. This ranking of evaluation results is displayed as a matrix of scenarios versus tactics called "traffic light". For every reasoning framework, *ArchE* sends *DescribeTactic* in parallel. Each reasoning framework provides *ArchE* with questions that describe suggested tactics (if applicable). For example, our reasoning framework would ask the user to apply the tactic of splitting on a particular responsibility, in order to satisfy a modifiability scenario.
5. *ArchE* shows to the user all the questions sent by reasoning frameworks. The cycle goes back to step 1.

3.2 Governing Reasoning Frameworks

When implementing the *ArchE-RF Interface*, a reasoning framework is expected to support six basic functionalities, which will hook into the search cycle described above. The functionalities are:

- Self Description (manifesto)
- Creating Initial Design
- Analyzing (for commands *Analyze* and *AnalyzeAndSuggest*),
- Suggesting Tactics (for command *AnalyzeAndSuggest*)
- Applying Tactics (for commands *ApplyTactic* and *ApplySuggestedTactic*)
- Describing Tactics (for command *DescribeTactic*)

ArchE does not require a reasoning framework to implement all the functionalities, but at least *Self Description* must be implemented to enable communication with *ArchE*. The implementation of the remaining functionalities is up to the researcher, depending on the type of reasoning framework wanted. The *Analyzing* functionality is generally present in any reasoning framework. For example, if we build our modifiability reasoning framework just to apply CIA on the module view, we can implement the *Analyzing* and *Creating Initial Design* parts and ignore other functionalities. However, if we would like our reasoning framework to be able to alter the architecture (after performing analysis), then we also need to implement the functionalities of *Suggesting Tactics*, *Applying Tactics* and *Describing Tactics*.

In addition to a command-based interface for interacting with *ArchE*, the *ArchE-RF Interface API* provides guidelines to implement the reasoning framework internals. These guidelines can be seen as a small object-oriented framework [11] that predefines the overall design of a plugin, its decomposition into Java interfaces and classes, the main methods to be overridden, and the general flow of control derived from the interaction commands. These features significantly reduce the design decisions that have to be made by a researcher when creating plugins for *ArchE*.

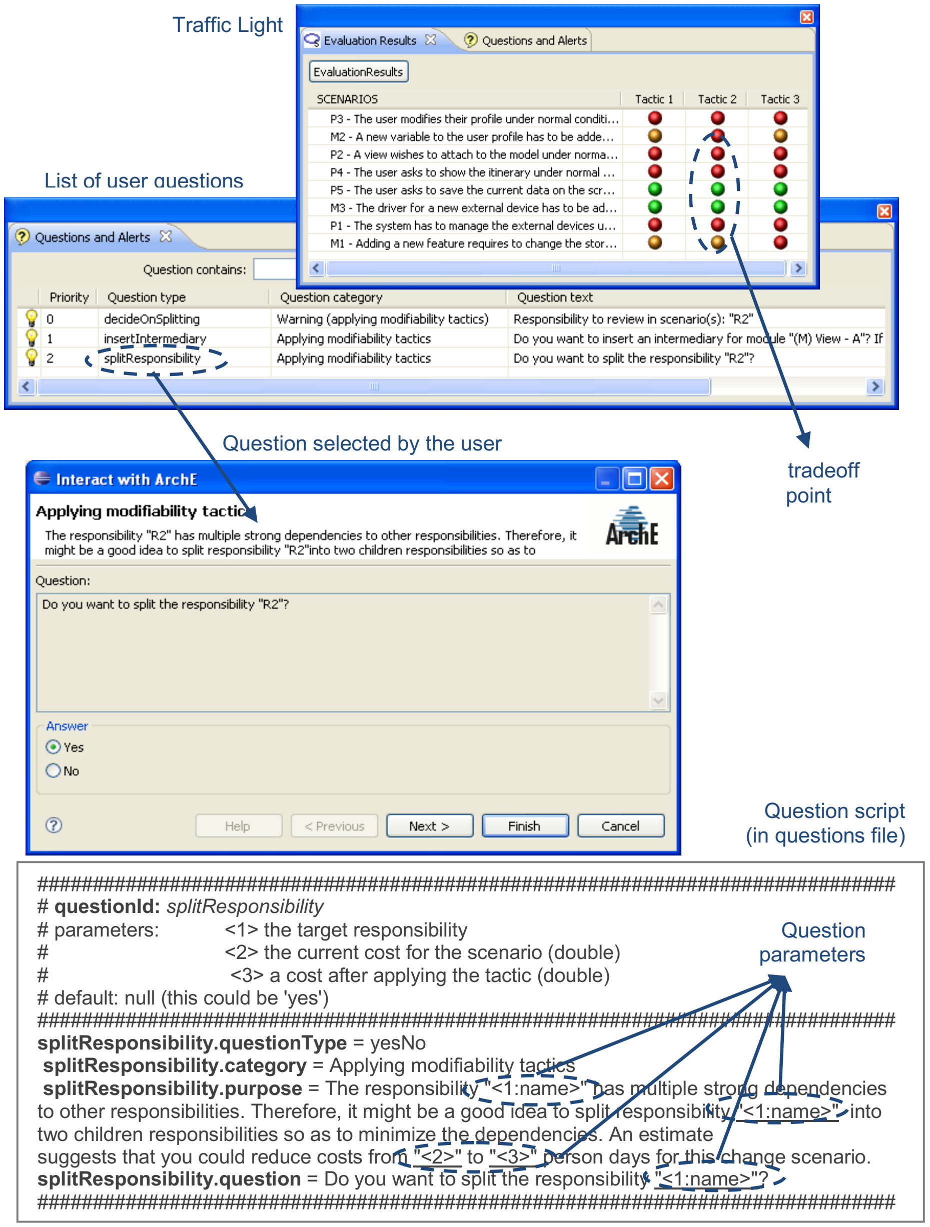
The *ArchE-RF Interface API* is structured into four layers. Each layer provides services for the upper layers, although there is no strict layering.

- *Communication layer.* It is the top-level layer that includes all the classes and interfaces related to interacting with *ArchE* via the *XmlBlaster*. It provides functionalities such as: registration of a reasoning framework with *ArchE* at runtime; reception of an interaction command from the *XmlBlaster* and delegation of its execution to the *Execution* layer; communication of progress messages and notice of command cancellations.
- *Execution layer.* It is equipped with a set of algorithms, each processing a different interaction command as forwarded from the *Communication* layer. Based on the services from the two layers below it, the *Execution* layer provides functionalities such as: restoring, saving and deletion of the architecture in the *ArchE Database*; exception handling, etc.
- *Reasoning Framework layer.* It provides the *ArchEReasoningFramework* class, which has to be extended by a researcher in order to implement a specific reasoning framework. It also provides other helping classes that she may use to handle inputs and outputs for an interaction command.
- *Data layer.* It is the bottom-level layer that provides the upper layers with the concepts shown in Figure 1. It includes the Java interfaces needed to manage the key concepts, which must be mapped to concrete classes and database tables.

3.3 Interaction with the User

The user gets to know about the reasoning framework proposals for the current architecture through two GUI mechanisms: the “traffic-light” metaphor and the user questions. Figure 5 shows a traffic light snapshot for modifiability and performance scenarios, along with potential scenario improvements when applying different tactics. The columns display color-coded ball icons that represent the tactics being evaluated by *ArchE*. A green ball indicates that the scenario will be satisfied if that tactic is applied, while a red ball indicates that the scenario will not be satisfied. Note also how the effects of the tactics on the scenarios lead to quality-attribute tradeoffs.

The snapshot below the traffic light shows a list of user questions. Typically, a question describes the purpose of a particular tactic. For instance, Figure 5 displays a question dialog for the tactic of splitting a costly responsibility. If the user enters a positive answer, then *ArchE* will trigger the corresponding architectural transformation. The types of questions associated to a reasoning framework must be specified by the reasoning framework developer in a questions file that supplements the manifesto. This questions file let *ArchE* know about the template and parameters of each possible question. The bottom part of Figure 5 shows how the question scripts look like. When the *ArchE* engine invokes the *DescribeTactic* command and the reasoning framework returns a question instance, *ArchE* loads its associated template and substitutes the placeholders of the text with specific question parameters. The *ArchE* GUI uses that information to display the question by means of predefined graphical widgets. Once the user picks and answers a particular question, *ArchE* translates the results into an *ApplyTactic* command for the reasoning framework that provided that question.



SCENARIOS	Tactic 1	Tactic 2	Tactic 3
P3 - The user modifies their profile under normal conditi...	🔴	🔴	🔴
M2 - A new variable to the user profile has to be adde...	🟠	🔴	🟠
P2 - A view wishes to attach to the model under norma...	🔴	🔴	🔴
P4 - The user asks to show the itinerary under normal ...	🔴	🔴	🔴
P5 - The user asks to save the current data on the scr...	🟢	🟢	🟢
M3 - The driver for a new external device has to be ad...	🟢	🟢	🟢
P1 - The system has to manage the external devices u...	🔴	🔴	🔴
M1 - Adding a new feature requires to change the stor...	🟠	🟠	🔴

Question contains:

Priority	Question type	Question category	Question text
0	decideOnSplitting	Warning (applying modifiability tactics)	Responsibility to review in scenario(s): "R2"
1	insertIntermediary	Applying modifiability tactics	Do you want to insert an intermediary for module "(M) View - A"? If
2	splitResponsibility	Applying modifiability tactics	Do you want to split the responsibility "R2"?

Applying modifiability tactic

The responsibility "R2" has multiple strong dependencies to other responsibilities. Therefore, it might be a good idea to split responsibility "R2"into two children responsibilities so as to

Question:

Do you want to split the responsibility "R2"?

Answer

[✓ Yes] No

```
##################################################################################
# questionId: splitResponsibility
# parameters:          <1> the target responsibility
#                      <2> the current cost for the scenario (double)
#                       <3> a cost after applying the tactic (double)
# default: null (this could be 'yes')
##################################################################################
splitResponsibility.questionType = yesNo
 splitResponsibility.category = Applying modifiability tactics
 splitResponsibility.purpose = The responsibility "<1:name>" has multiple strong dependencies
to other responsibilities. Therefore, it might be a good idea to split responsibility "<1:name>" into
two children responsibilities so as to minimize the dependencies. An estimate
suggests that you could reduce costs from "<2>" to "<3>" person days for this change scenario.
splitResponsibility.question = Do you want to split the responsibility "<1:name>"?
##################################################################################
```

Fig. 5. Configuration and visualization of tactics in *ArchE*

4 Implemented Reasoning Frameworks and Lessons Learned

Currently, we have created two reasoning framework plugins using the *ArchE-RF Interface*. The first plugin is a full-fledged reasoning framework for modifiability (as outlined in sub-section 2.1), which served to test and tune the infrastructure. The second plugin is a reasoning framework for real-time performance that takes advantage of an existing analytic solver called MAST[5]. MAST [12, 15] is a toolset for describing event-driven real-time systems and performing schedulability analysis. Figure 6 shows a snapshot of *ArchE* running the two plugins. In general, validating reasoning frameworks with respect to the scope and accuracy of their predictions is the job of the reasoning framework developer and not a portion of *ArchE*.

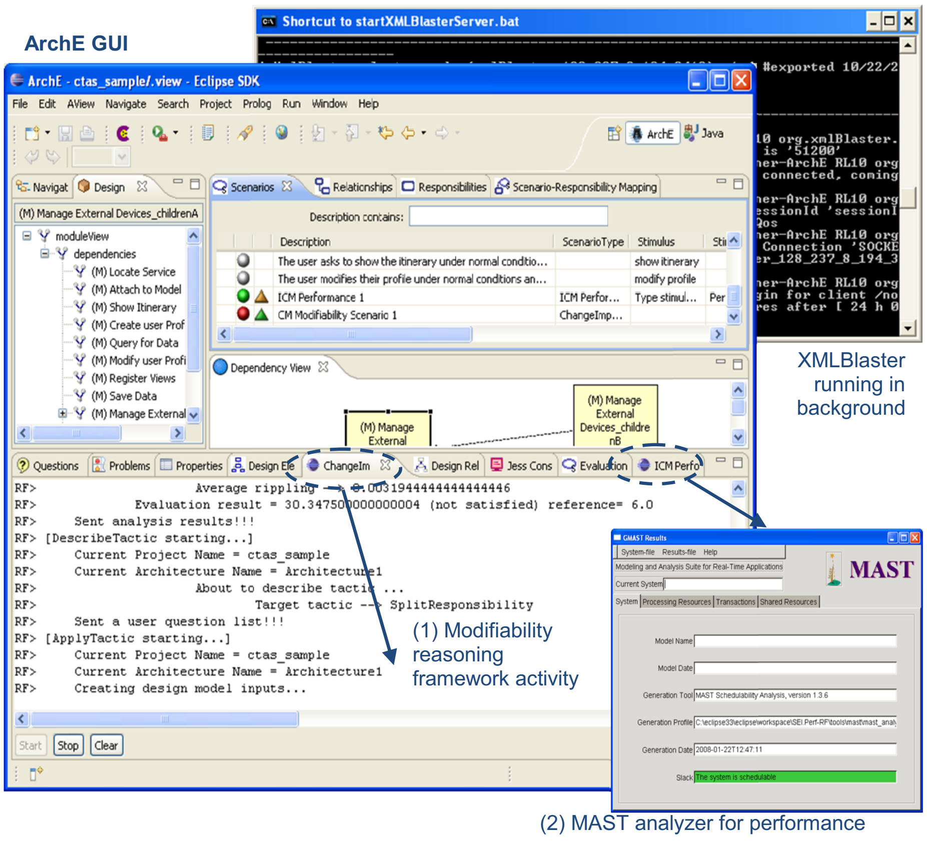
After writing its manifesto, the modifiability reasoning framework was implemented from scratch in Java. Initially, we defined subclasses for the responsibility dependencies and the responsibility structure. We also created a class to represent the module view. Then, we implemented a subclass of the *ArchEReasoningFramework* class that encapsulates the interpretation and the formulae for computing various metrics such as cost, coupling and cohesion. On this basis, we codified rules that looked at the values of these metrics to configure possible tactics. Finally, we equipped the reasoning framework with architectural transformations for the tactics, and we also wrote the corresponding questions file.

The performance reasoning framework was conceived as an "analyzer" with no support for tactics. The implementation steps were similar to the ones carried out for the modifiability plugin, except that we wrapped the MAST solver to supply the *Analyze* functionality. The MAST input is an ASCII file that consists of an arrangement of tasks with timing requirements (e.g., latency) and events linking the tasks. A worst-case analyzer processes this specification and outputs the timing behavior of the system. In our *ArchEReasoningFramework* subclass, the *Analyze* implementation converts the performance scenarios and their responsibilities to tasks, considering the responsibility relationships as event reactions between tasks. The task model is sent to a file and fed into the MAST toolset. The worst-case latency results are then compared against the timing requirements to determine the schedulability of the scenarios. We are now working on the addition of a set of performance tactics to this plugin.

The reliance of *ArchE* on reasoning frameworks favors integrability and modular reasoning about quality attributes. One of the research questions here is the extent to which the interactions (i.e., dependencies) among quality-attribute models can be reduced. The implementations above shed light on general issues about these interactions and also exposed some drawbacks of the blackboard approach.

In the current design, dataflow interactions arise because the reasoning frameworks often share (parts of) the architectural representation (e.g., responsibilities, elements of architectural views). Anyway, this architectural representation must be kept consistent at all times. Our plugins shared responsibilities but worked on separate architectural views (i.e., a module view and a task view respectively), and only the modifiability plugin had the capability of modifying the architecture. Because of these factors, the consistency checking was relatively simple. For instance, if a modifiability tactic splits a responsibility that appears in a performance scenario, then the performance reasoning framework is asked to update its task model and run the schedulability analysis again. We believe that a general treatment of opportunistic or harmful types of interactions would require more knowledge about the architectural representation, the effects of tactics or the user's inputs.

[5] MAST homepage: http://mast.unican.es/


ArchE GUI
Shortcut to startXMLBlasterServer.bat
ArchE - ctas_sample/.view - Eclipse SDK
File Edit AView Navigate Search Project Prolog Run Window Help
ArchE
Java
Navigat
Design
Scenarios
Relationships
Responsibilities
Scenario-Responsibility Mapping
Dependency View
(M) Manage External
(M) Manage External Devices_childrenB
Questions
Problems
Properties
Design Ele
ChangeIm
Design Rel
Jess Cons
Evaluation
ICM Perfo
Start
Stop
Clear
XMLBlaster running in background
(1) Modifiability reasoning framework activity
(2) MAST analyzer for performance
MAST

(M) Manage External Devices_childrenA

- moduleView
  - dependencies
    - (M) Locate Service
    - (M) Attach to Model
    - (M) Show Itinerary
    - (M) Create user Prof
    - (M) Query for Data
    - (M) Modify user Profi
    - (M) Register Views
    - (M) Save Data
    - (M) Manage External

Description contains:

		Description	ScenarioType	Stimulus	Sti
●		The user asks to show the itinerary under normal conditio…		show itinerary	
●		The user modifies their profile under normal conditions an…		modify profile	
🟢	▲	ICM Performance 1	ICM Perfor…	Type stimul…	Per
🔴	▲	CM Modifiability Scenario 1	ChangeImp…		

```
#exported 10/22/2
10 org.xmlBlaster.
 is '51200'
ner-ArchE RL10 org
 connected, coming
ner-ArchE RL10 org
essionId 'sessionI
Qos
ner-ArchE RL10 org
 Connection 'SOCKE
er_128_237_8_194_3
ner-ArchE RL10 org
gin for client /no
res after [ 24 h 0
```

```
RF>                    Average rippling --> 0.0031944444444444446
RF>                Evaluation result = 30.347500000000004 (not satisfied) reference= 6.0
RF>     Sent analysis results!!!
RF> [DescribeTactic starting...]
RF>     Current Project Name = ctas_sample
RF>     Current Architecture Name = Architecture1
RF>                    About to describe tactic ...
RF>                            Target tactic --> SplitResponsibility
RF>     Sent a user question list!!!
RF> [ApplyTactic starting...]
RF>     Current Project Name = ctas_sample
RF>     Current Architecture Name = Architecture1
RF>     Creating design model inputs...
```

GMAST Results

System-file Results-file Help

Modeling and Analysis Suite for Real-Time Applications

Current System

System | Processing Resources | Transactions | Shared Resources

Model Name	
Model Date	
Generation Tool	MAST Schedulability Analysis, version 1.3.6
Generation Profile	C:\eclipse33\eclipse\workspace\SEI.Perf-RF\tools\mast\mast_analy
Generation Date	2008-01-22T12:47:11
Slack	The system is schedulable

Fig. 6. The *ArchE* prototype executing two reasoning frameworks as plugins

The management of tradeoffs is decoupled into two aspects. The first aspect has to do with the "traffic light" metaphor, so that the user must decide on a tactic making a quality-attribute balance that is good enough for her scenarios of interest. The second aspect comes from the opportunistic/harmful interactions discussed above. A simple source of tradeoffs is the parameters of responsibilities [2]. For instance, when inserting an intermediary due to modifiability reasons, the modifiability reasoning framework can impose a minimum execution time for that responsibility, but this constraint on the execution time parameter later impacts on the schedulability analysis of the performance reasoning framework. Putting mechanisms in place for *ArchE* to support this second aspect of trade-offs is a topic for further research.

Regarding search, each reasoning framework looks locally for tactics that change the architectural structure. However, the resulting architectural transformations do not always guarantee an improvement of the evaluation function, because that function depends on both the architectural configuration and tactic-specific parameters. For instance, when applying the tactic of splitting a responsibility, we must set the costs for the children responsibilities and set the rippling probabilities for their dependencies. Different choices for these values lead to different instances of the same tactic, some of which reduce the cost of the change and some others do not. The problem of finding an adequate configuration of values for a given tactic is not trivial, and it often needs heuristic search.

We additionally observed some side-effects of the blackboard architecture on usability. A first issue is the processing overhead forced by the main control strategy, because the *ArchE* engine does not know the semantics of the user's actions. A second issue (related to the control strategy) is that the reasoning framework activities for responding to the *ArchE* commands have limited visibility through the GUI. Therefore, while *ArchE* is running, the user can only handle or inspect reasoning framework features at specific points of the exploration process. Future developments should provide a more flexible user-interaction schema.

5 Related Work

The analysis of component-based systems by applying quality-attribute techniques has been an active field of research and technology transfer for many years. Several quality-specific approaches have been developed [7, 10, 14, 15], although few of them have tackled the integration of models and analysis tools. To begin with, the Predictable Assembly from Certifiable Components (PACC) initiative at the SEI has focused on building component-based systems that have predictable behaviors prior to implementation [15]. PACC uses the notion of reasoning frameworks in combination with model checking to analyze performance and safety properties but also to enforce the assumptions required by each analysis technique when applied to the systems. This technology can be applied to predict other properties as well (e.g., reliability, security). As evidenced by the MAST example, we think these techniques can be integrated into *ArchE* with little effort.

The DeSiX approach [7] provides tools for component-based systems on multiprocessor architectures that allow for design space exploration. Here, scenario-based analyses for performance, reliability and cost serve to focus the design on particular architectural configurations. The developer can map usage profiles to simulation tasks, and then visualize the resulting architectures using Pareto curves. When compared to *ArchE*, a drawback of DeSiX is that it does not support automated search, and the developer manually selects configurations to be evaluated by the tool.

Other researchers have proposed a view of software engineering as a search problem [9], in which automation is supported by optimization techniques. Along this line, Grunske [13] has investigated the integration of quality-attribute techniques using genetic algorithms for some experiments involving reliability and cost requirements. Also, he has proposed a generic model for quality-attribute evaluation [14] that contains four elements, namely: encapsulated evaluation models, composition algorithms for these evaluation models, operational/usage profiles, and evaluation algorithms to determine relevant quality measures from the evaluation models. This perspective is similar in spirit to that of reasoning framework, although it does not consider explicitly the aspect of architectural transformations. Nonetheless, Grunske has pointed out challenges for the combined use of quality-attribute models and tool support, such as composability, analyzability and complexity issues.

More recently, Edwards et al. have [10] coined the term "model interpreter" as a vehicle to transform component-based models into analysis models by means of model-driven engineering (MDE) techniques. Consequently, they have developed a "tool-chain" called XTEAM that supports and integrates different types of model interpreters. These interpreters are able to implement transformations between high-level component models (amenable to architectural reasoning) and low-level analysis models (amenable to prediction of component assembly properties). This approach is still experimental and has many analogies with the PACC work, but unlike *ArchE*, it does not seem to focus on the exploration of the design space.

6 Conclusions

In this paper, we have described a tool approach for incentivizing the use of quality-attribute models in architectural design. The *ArchE* approach relies on having a collection of reasoning frameworks that are each specialized for a single quality attribute but that work together in the creation and analysis of architectural designs. *ArchE* is not intended to perform an exhaustive or optimal search in the design space; rather, it is an assistant to the architect that can point out "good directions" in that space. Along this line, the contributions of this work are the encapsulation of quality-attribute knowledge and the tool infrastructure to accommodate that knowledge.

The *ArchE-RF Interface* constitutes an important step towards improving the design of the *ArchE* prototype. Nonetheless, there are issues that need further discussion and implementation efforts. Some of these issues are:

- Incorporation of UML features for architectural modeling, and linking *ArchE* to other development tools.
- Management of tradeoffs between solutions proposed by individual reasoning frameworks, under multiple criteria (e.g., cost, utility, uncertainty).
- Experiments with searching techniques and more powerful solvers (e.g., simulated annealing, planning, SAT, etc.).
- Support for recording design decisions, as an extension of quality-attribute analysis results and tactic proposals.

Finally, we believe that the more reasoning frameworks that are available, the broader the reasoning capabilities of *ArchE* will be. Thus, we hope this work will stimulate researchers, educators and practitioners to plug in and share analysis/design models for various quality attributes, in order to foster architecture-centric practices.

References

1. Bachmann, F., Bass, L., Klein, M., Shelton, C.: Experience Using an Expert System to Assist an Architect in Designing for Modifiability. In: Proceedings 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), Oslo, Norway, p. 281 (2004)
2. Bachmann, F., Bass, L., Klein, M., Shelton, C.: Designing Software Architectures to Achieve Quality Attribute Requirements. Software IEE 152(4), 153–165 (2005)
3. Bachmann, F., Bass, L., Nord, R.: Modifiability Tactics. Technical report CMU/SEI-2007-TR-002. Software Engineering Institute, Pittsburgh, PA (2007)

4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Reading (2003)
5. Bass, L., Ivers, I., Klein, M., Merson, P., Wallnau, K.: Encapsulating Quality Attribute Knowledge. In: Proceedings 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005), Pittsburgh, PA, pp. 193–194. IEEE Computer Society, Los Alamitos (2005)
6. Bohner, S., Arnold, R.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos (1996)
7. Bondarev, E., Chaudron, M., de With, P.: Quality-Oriented Design Space Exploration for Component-Based Architectures. Computer Science Report. University of Technology, Eindhoven, The Netherlands (2006)
8. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture. A System of Patterns. John Wiley & Sons, Chichester (1996)
9. Clarke, J., Dolado, J., Harman, M., Hierons, R., Jones, R., Lumkinm, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., Shepperd, M.: Reformulating Software Engineering as a Search Problem. Software IEE 150(3), 161–175 (2003)
10. Edwards, G., Seo, C., Medvidovic, N.: Construction of Analytic Frameworks for Component-Based Architectures. In: Proceedings of the Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS). Campinas, Sao Paulo, Brazil (2007)
11. Fayad, M., Schmidt, D., Johnson, R. (eds.): Building Application Frameworks: Object-Oriented Foundations of Framework Design. Wiley, Chichester (1999)
12. Gonzalez Harbour, M., Gutierrez García, J.J., Palencia Gutiérrez, J.C., Drake Moyano, J.M.: MAST: Modeling and Analysis Suite for Real Time Applications. In: Proceedings 13th Euromicro Conference on Real-Time Systems (ECRTS), IEEE Comp. Society, Washington (2001)
13. Grunske, L.: Identifying "Good" Architectural Design Alternatives with Multi-Objective Optimization Strategies. In: International Conference on Software Engineering (ICSE), Workshop on Emerging Results, pp. 20–28, 849–852. ACM Shanghai, China (2006)
14. Grunske, L.: Early quality prediction of component-based systems - A generic framework. Journal of Systems and Software 80(5), 678–686 (2007)
15. Ivers, J., Moreno, G.A.: Model-driven development with predictable quality. In: Companion 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications Companion (OOPSLA 2007), Montreal, Quebec, Canada (2007)
16. McGregor, J., Bachmann, F., Bass, L., Bianco, P., Klein, M.: Using an Architecture Reasoning Tool to Teach Software Architecture. In: Proceedings 20th Conference on Software Engineering Education & Training (CSEE&T 2007), pp. 275–282. IEEE Computer Society, Los Alamitos (2007)
17. Wilkins, D., des Jardins, M.: A Call for Knowledge-based Planning. AI Magazine 22(1) (Spring, 2001)
18. Wirfs-Brock, R., McKean, A.: Object Design: Roles, Responsibilities, and Collaborations. Addison-Wesley, Boston (2003)