Elements of a Usability Reasoning Framework

Jinhee Lee
Len Bass

September 2005

Software Architecture Technology Initiative

Technical Note
CMU/SEI-2005-TN-030

Contents

List of Figures

List of Tables

Abstract

This technical note brings together two different threads of work: (1) investigating the relationship between usability and software architecture that has generated a number of usability scenarios with implications for software architecture and (2) developing an architecture design assistant, Architecture Expert (ArchE). One key element of ArchE is that quality attribute knowledge can be encapsulated into *reasoning frameworks*, and a Carnegie Mellon University Master of Software Engineering project team has developed an ArchE reasoning language (ARL) with which to specify the actions of reasoning frameworks within ArchE.

This note describes an ARL implementation of two usability scenarios: (1) displaying progress feedback and (2) allowing cancel. These implementations begin to provide ArchE with the ability to reason about aspects of usability that have software architecture implications.

1 Introduction

Earlier this year, we described the encapsulation of quality attribute knowledge into *reasoning frameworks* [Bass 05]. A reasoning framework embodies the information necessary to interpret an architecture in terms of a quality attribute theory. We described two reasoning frameworks for real-time scheduling and modifiability based on rate monotonic analysis and impact analysis [Bachmann 05]. This work was done while developing an architecture design assistant (ArchE) whose preliminary design we described [Bachmann 03] and which was implemented in the Jess rule language [Friedman-Hill 03]. In addition to these published descriptions of our work, we have also been working to make reasoning frameworks easier to incorporate into ArchE. In conjunction with a team in the 2005 Master of Software Engineering program, we defined an ArchE reasoning language (ARL) and a set of associated tools that allow architects to specify reasoning frameworks and incorporate them within ArchE.

In this technical note, we describe how this machinery is applied to create the initial elements of a usability reasoning framework. At the time this technical note was published, ArchE was totally functional, but the ARL compiler was not.

This technical note is organized as follows:

- Section 2 describes the usability quality architecture theory upon which the reasoning framework is based.
- Section 3 describes the machinery we created—that is, the ARL—and how something specified in the ARL is incorporated into ArchE.
- Section 4 describes the process we used to define the reasoning frameworks, beginning with the quality attribute theories.
- Sections 5 and 6 include the specification of two elements of the reasoning framework that support the usability scenarios of *progress feedback* and *cancellation*.

2 Usability and Software Architecture

A very successful technique used by software architecture designers to support usability is separating presentation (input and output) from the rest of the application. The most widely used architectural pattern for accomplishing this separation is Model View Controller (MVC), which was developed in the early 1980s and documented by Buschmann [Buschmann 96]). However, using MVC (or any other separation-based pattern) complicates concerns that cut across MVC component types—particularly those that involve view and model.

We investigated such cross-cutting concerns [Bass 01] and identified scenarios that represent usability concerns involving the view, controller, *and* model. For example, giving the user the ability to cancel a long-running command (an important usability feature) involves restoring the application and its resources to their state prior to the invocation of the command.

We packaged our scenarios with other information to create Usability-Supporting Architectural Patterns (USAPs) [John 04]. USAPs have five parts:

1. the scenario of interest (e.g., giving the user the ability to cancel commands)
2. constraints on applying the scenario (e.g., cancel only makes sense for long-running commands)
3. benefits to the user when this scenario is supported (e.g., supporting cancel makes the performance of routine tasks more efficient by providing the ability to recover from inadvertent slips)
4. general responsibilities that any implementation of the scenario must support (e.g., the system must always listen for a cancel command from the user)
5. a sample solution embedded in the MVC pattern (e.g., the controller listens for cancel commands)

General responsibilities are presented because even though MVC is the most widely used separation pattern, other patterns are available. General responsibilities show designers what they need to be concerned about without prejudging any portion of their design.

A sample solution is presented to demonstrate how particular responsibilities are distributed among elements of the pattern.

In the work described by this technical note, we use the scenario of interest to capture the requirement that the scenario be supported, and we use general responsibilities as (1) a means for the reasoning framework to determine whether the scenario is currently supported and (2) as a means of transforming the architecture (by adding responsibilities) to cause the architecture to support the scenario.

3 ARL and ArchE

The ARL is currently under development. Therefore, the ARL code in the appendices is not yet fully implemented. The language has stabilized, however, and the implemented code should not vary dramatically from the code presented here.

Figure 1 shows how the ARL is used to specify a reasoning framework that is then used to assist in the design process.

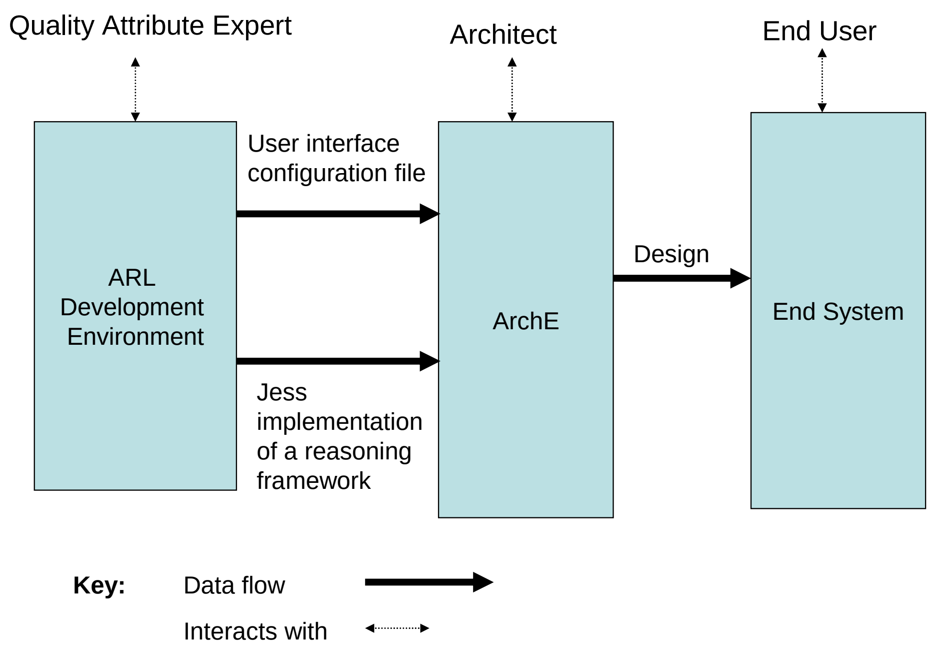

Quality Attribute Expert
Architect
End User
User interface configuration file
ARL Development Environment
ArchE
Design
End System
Jess implementation of a reasoning framework
Key:
Data flow
Interacts with

Figure 1: Information Flow from the Specification of a Reasoning Framework to an End User

First, the quality attribute expert uses the ARL development to prepare input for ArchE and specify the reasoning framework. The ARL development environment then outputs two forms of data that are incorporated into ArchE:

1. a configuration file used by the ArchE input/output component to allow the architect using ArchE to specify usability scenarios
2. a collection of Jess rules [Friedman-Hill 03] that represent the workings of the reasoning framework

   Jess rules are divided into four sections: (1) preparation to ensure that all parameters have been furnished, (2) interpretation to convert the current architectural specification to a quality attribute model, (3) evaluation to determine whether the model satisfies the quality attribute requirement, and (4) tactic generation to determine which architectural tactics [Bass 03, Ch. 5] will improve the architecture with respect to the quality attribute of interest.

Next, the architect interacts with ArchE to produce a design for a particular system. Finally, the end user interacts with the designed system.

Several levels of indirection complicate the specification of a reasoning framework. Consider progress feedback: the USAP for progress feedback specifies that if an interaction takes less than two seconds, no special interaction is required. If an interaction takes between 2 and 10 seconds, the cursor shape should be changed to indicate a busy state, and if the interaction takes more than 10 seconds, some indication of progress should be given. The questions become (1) how does the quality attribute expert specify this situation? and (2) where is the determination made regarding the type of feedback to present?

The quality attribute expert cannot make that determination because the system being designed is unknown to the expert. Therefore, the specification in the reasoning framework must consider the following information:

- The designer knows which commands will take more than 2 seconds and less than 10 seconds. In this case, the designer can specify that the cursor shape be changed for those commands.
- The designer does not know which commands will take more than 2 seconds and less than 10 seconds. In this case, the designer must specify that the designed system predict the elapsed time of a command and adjust the cursor shape appropriately.
- For commands that will take longer than 10 seconds, a reasonable assumption is that the designer will not know their duration, so the system must predict it.

Because the accuracy of the predicted duration has an impact on the type of progress feedback that should be given, the designer must consider it carefully. If a prediction is accurate to within 20%, it should be shown to the end user; if it is not accurate to within 20%, a form of progress feedback other than time should be used (e.g., percent complete or number of items processed).

4 Pieces of a Reasoning Framework and the Process of Developing a Reasoning Framework

We have previously defined the reasoning framework concept and described its core in a fashion useful for several different contexts, including ArchE [Bass 05]. Different contexts are accommodated by extending the core, and this technical note, which focuses on an ArchE reasoning framework, includes the ArchE-specific extensions only.

An ArchE reasoning framework has four sections:

1. preparation

   This section ensures that all information required for the interpretation is available. It does this by questioning the architect or acquiring and transforming architecture properties.

2. interpretation

   This section constructs the quality attribute model from information available in the current architecture, the current set of responsibilities, and the scenarios.

3. evaluation

   This section evaluates the quality attribute model to determine the response measure.

4. tactics

   This section suggests modifications to the current architecture, the current set of responsibilities, or the scenarios to improve the quality attribute response.

Creating an ArchE reasoning framework is a four-step process: (1) define the scenario, (2) organize responsibilities using a graph that shows the dependencies among the responsibilities as well as their sequence, (3) develop an English-language description of the sequence of activities (this serves as the specification of the ARL code), and (4) develop the code.

Sections 5 and 6 are organized to reflect this sequence of steps and show how the ArchE sections are realized by the progress feedback and cancellation frameworks.

5 Progress Feedback Reasoning Framework

In this section, we describe the implementation of the progress feedback reasoning framework. We follow the sequence described at the end of Section 4: focus on scenarios, organize responsibilities, derive an English description of the activities, and develop the code.

First, we present the general scenario for progress feedback. Next, we describe how the user interface (UI) of ArchE is modified so that the architect can specify the progress feedback scenario. Finally, we describe the ARL implementation, including general responsibilities from the progress feedback USAP and implementation structure and description. The code for the UI portion and the reasoning framework are presented in Appendices A and B.

5.1 General Scenarios

The base scenario for progress feedback is *The user initiates a long-running task*. The system provides feedback by (1) changing the cursor shape and (2) indicating how much of the task has been accomplished and how much remains.

Each reasoning framework is triggered by some set of scenarios. The scenario's type categorizes it so that the framework can be triggered by its existence or by a change in one of its implied responsibilities. We currently view the progress feedback reasoning framework as independent and define its type as *progress feedback*. It is also possible that the progress feedback is part of a larger usability reasoning framework of the type *usability*.

The architect must input the scenario in a structured form because ArchE does not use any natural language-parsing techniques. We used the six-part scenario formulation [Bass 03, Ch. 4] included in Table 1. Initially, we use a response measure of whether the scenario is satisfied. A response measure of partial success (i.e., the number of responsibilities satisfied) could also be used, and we explore that concept for the cancellation reasoning framework.

Table 1: Six-Part Scenario Formulation for Progress Feedback

Element	**Description**
Stimulus	Long-running task is initiated (e.g., download file from a Web site).
Stimulus Source	End user
Environment	At runtime
Artifact	System
Responses	• System shows progress indicator to the end user with estimated time to complete or with information about the current operations. • Calculate the number of operations and show the current operations, the number of completed operations, and the number of operations to be done. • System changes the cursor shape.
Response Measure	Boolean (Yes/No) • Does the system contain the functionality of showing the time to complete? • Does the system provide an explanation about operations on which the system is working? • Does the system provide the number of completed operations and operations to be done? • Does the system change the cursor shape appropriately?

The UI that the architect uses to input the scenario is described in the next section.

5.2 Create the UI Configuration File for ArchE

ArchE derives the following information from a configuration file named *rfconfig.xml*:

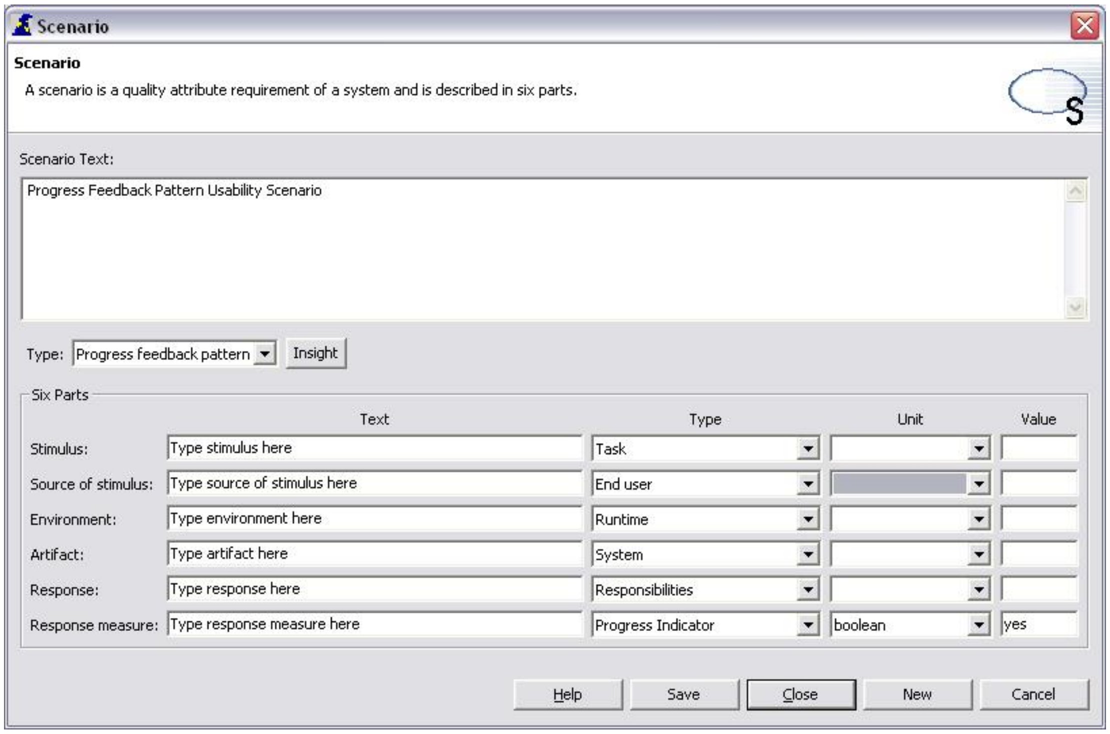
- scenario type
- relationship type
- parameter type
- model elements

The *rfconfig.xml* file should be created using the ArchE Configurator tool. Each element in the general scenario can have specific types and units if the elements are used in the reasoning framework. For this progress feedback reasoning framework, the following elements may have some limited types and units as shown in Table 2.

Table 2: Element Types and Units for Progress Feedback

Element	**Type**	**Units**	**Default Type**	**Default Value**
Stimulus	Task	N/A	Task	N/A
Stimulus source	End user, system	N/A	End user	N/A
Environment	Runtime	N/A	Runtime	N/A
Artifact	System	N/A	System	N/A
Responses	Responsibilities	N/A	Responsibilities	N/A
Response measure	Progress indicator, cursor shape	Boolean (yes/no)	Progress indicator	Yes

If the new *rfconfig.xml* file is loaded by ArchE, the scenario wizard displays the screen shown in Figure 2 to users.


Scenario

Scenario

A scenario is a quality attribute requirement of a system and is described in six parts.

Scenario Text:

Progress Feedback Pattern Usability Scenario

Type: Progress feedback pattern

Six Parts

	Text	Type	Unit	Value
Stimulus:	Type stimulus here	Task		
Source of stimulus:	Type source of stimulus here	End user		
Environment:	Type environment here	Runtime		
Artifact:	Type artifact here	System		
Response:	Type response here	Responsibilities		
Response measure:	Type response measure here	Progress Indicator	boolean	yes

Figure 2: Creating a Concrete Scenario in ArchE

For the complete *rfconfig.xml* file for this reasoning framework, see Appendix B.

5.3 Responsibilities

The progress feedback USAP contains 14 general responsibilities based on the following criteria:

- If the initiated task takes between 2 and 10 seconds, the cursor shape should be changed.
- If the initiated task takes longer than 10 seconds, a progress indicator should be displayed.
- If the estimated time remaining is accurate to within 20%, the progress indicator should display the time remaining.
- If it is not, the indicator should show the number of items processed, number left to process, % completion, and so forth.

A progress indicator must be updated periodically.

As shown in Table 3, 14 general responsibilities implement the progress feedback.

Table 3: General Responsibilities for Progress Feedback

Responsibility	**Description**
R01	Estimate elapsed time of the task.
R02	Determine the type of progress indicator.
R03	Change the cursor shape to busy.
R04	Calculate the accuracy of any time estimate.
R05	Show time-based progress feedback.
R06	Show non-time-based progress feedback.
R07	Show time remaining to complete.
R08	Calculate and display the number of operations to be done.
R09	Show information about the current working operation.
R10	Update progress information periodically.
R11	Change the cursor to a normal indicator.
R12	Leave the progress dialog and show the completion of the task.
R13	Hide the progress dialog if the task is completed.
R14	Detect that the task is complete.

5.4 Workflow Description

5.4.1 Background

When an architect specifies a progress feedback type for a scenario, one of the scenario elements is the task for which progress feedback must be provided. This task translates into a responsibility within ArchE and, subsequently, is assigned to various elements of the architecture. We assume that the architect determines which tasks are long running, but an alternative to be explored assumes that ArchE determines long-running tasks.

The primary function of the progress feedback reasoning framework is to manipulate the responsibilities within ArchE. This manipulation consists of adding the necessary responsibilities (or ensuring that they are included). How the manipulated responsibilities are assigned to architectural elements is outside of the framework's scope.

In addition to manipulating responsibilities, the progress feedback reasoning framework also generates (in certain cases) a performance scenario for updating the display within a certain time period.

5.4.2 Identifying Unknowns

During preparation, any possible unknowns that affect the quality attribute model must be addressed. For the progress feedback reasoning framework, the possible unknowns are

1. Has a responsibility been linked to the task described in the scenario?

   If not, the architect must be prompted to make that linkage.

2. Is the duration of the task known, or does the system compute it? If it's the latter, is the accuracy of the estimate known?
3. Is progress feedback necessary?

   If it is, it must be updated periodically, and the update period must be determined.

5.4.3 Construct the Quality Attribute Model

The next step in a reasoning framework is to construct the quality attribute model. For the progress feedback reasoning framework, the quality attribute model uses the answers to the timing questions to determine whether all responsibilities are already included. If they are, the evaluation says that the scenario is satisfied. If they are not, the tactic section proposes—or in this case, performs—adding the necessary responsibilities to the responsibility graph within ArchE.

5.5 Implementation

This section elaborates the basic workflow described in Section 5.4. A reasoning framework is triggered whenever something (the adjustment of a responsibility, the addition of a new scenario, or the response to one of the questions that affects it) is changed. We describe the activities as if they are executed sequentially. However, they may not occur sequentially: their order may be determined by changes in the state of the design or requirements.

The reasoning framework operates primarily on the responsibility graph, and responsibilities within ArchE are maintained as a directed acyclic graph. Figure 3 shows the responsibility graph before the actions of the reasoning framework, and Figure 4 shows the graph after the additional responsibilities associated with progress feedback are added.

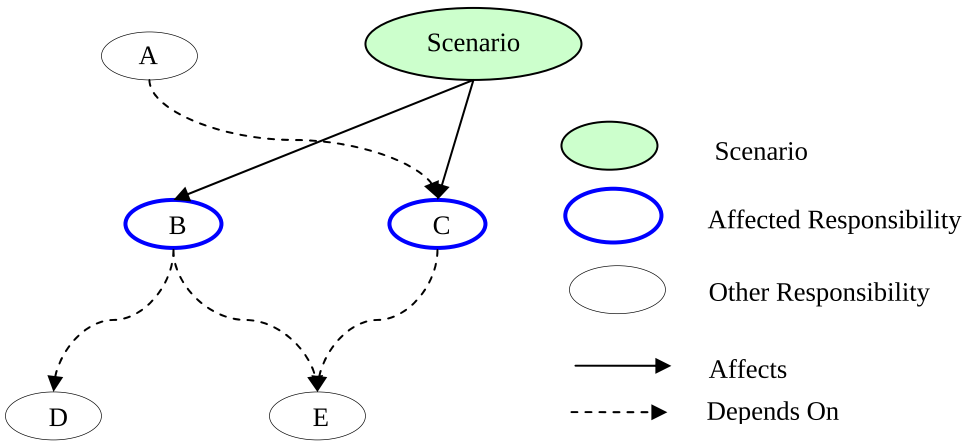


Figure 3: Responsibility Graph Linked to Scenario Before the Actions of the Reasoning Framework

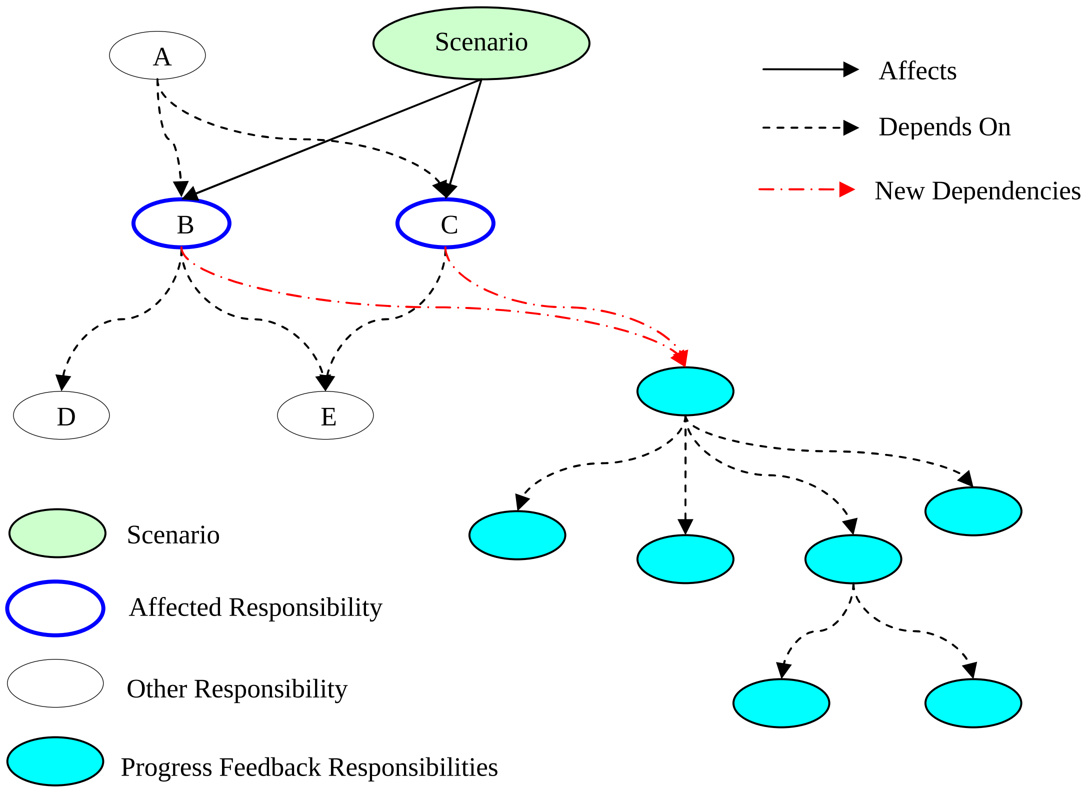


Figure 4: Responsibility Graph After Adding Progress Feedback Responsibilities

The scenario is assumed to be connected to some responsibilities that act as a root for the new responsibilities associated with the progress feedback.

The progress feedback reasoning framework will not remove or modify any current responsibilities, but it will link new ones to those responsibilities affected by the scenario. Also, even though a scenario affects a responsibility that has children, the progress feedback reasoning framework does not affect those children.

5.5.1 Preparation

During preparation, the reasoning framework determines if it has the information necessary to proceed. Activities flow as follows:

1. Retrieve the task and relevant responsibilities from the usability scenario.
    - Several responsibilities of the target system might be affected by the task that is a stimulus of the scenario. The progress feedback reasoning framework gets the responsibilities that are affected by the task defined in the usability scenario. If the scenario is not linked to any responsibilities, the architect is queried to determine this linkage.
2. Determine whether the system will have the functionality needed to estimate the elapsed time to complete the initiated task.
    - The designer can determine whether the system must perform the estimation.
    - If the system estimates, it will have additional responsibilities for estimating the elapsed time and calculating its accuracy. If the system does not estimate, no additional responsibilities are required, but a simple non-time-based progress indicator will be inserted into the design.
3. Determine who will provide the functionality to calculate the accuracy of the elapsed time and if the accuracy will be within 20%.
    - The designer determines whether the system will have the functionality to calculate accuracy.
    - The designer can guarantee that the system will estimate the execution time with good accuracy.
    - For example, if the system will not provide the functionality but the accuracy is guaranteed by the designer, the system may have a time-based progress bar.
4. Determine who will decide whether the progress feedback dialog is left visible after a long-running task is completed.
    - The system can show that the task is done and leave the progress dialog displayed.
    - The designer can determine whether to show the dialog when the task is complete.
    - The end user can specify (e.g., through a checkbox) that the dialog box is removed (or kept) when the task is complete.

5. Create two parameters: (1) P_EstimatedExecutionTime and (2) P_Accuracy.
    - Two parameters apply to subsequent steps: (1) elapsed time and (2) its accuracy.
    - The responsibilities of the progress USAP are the owners of the two parameters.
    - These two parameters are generated by estimation responsibilities as output.
    - For example, the R06 responsibility (show non-time-based progress feedback) will be executed only when the elapsed time is greater than 10 seconds and its accuracy is worse than 20%. All decisions to execute the responsibility will depend on these two parameters.

5.5.2 Instantiation and Evaluation

Instantiation refers to creating a quality attribute model from the architecture. The quality attribute model for progress feedback has no parameters that depend on the elements of the architecture design. The parameters of that model depend on the responsibilities in the responsibility graph instead of the design elements. Evaluating the quality attribute model, therefore, consists of verifying that the required responsibilities are included in the responsibility graph. If they are, the scenario is satisfied; if they are not, they are added to the responsibility graph as described below.

5.5.3 Apply Tactics

In this section, we describe how the responsibility graph is modified to reflect the progress feedback USAP. The general form was given in Figure 2 on page 10. The task considered in the scenario is linked to one or more responsibilities (parent responsibilities) in the responsibility graph. We identify three cases and distinguish them based on the architect's responses to the preparation questions. We assume that the architect will know (1) whether the elapsed time is less than 10 seconds and (2) the accuracy of any calculated estimate greater than 10 seconds.

1. The elapsed time for the task is between 2 and 10 seconds as specified by the architect.

    In this case, we add two responsibilities to the parent responsibilities: (1) R03 (change the cursor shape to busy) and (2) R11 (change the cursor to a normal indicator).

2. The elapsed time for the task will be greater than 10 seconds, and the remaining time can be estimated within an accuracy of 20%.

   In this case, the responsibility graph is modified to contain the responsibilities shown in Figure 5. The R10 responsibility (update progress information periodically) is accompanied by the generation of a performance scenario that has a stimulus of the period for updating the progress information (not identified as a separate responsibility) and a response of updating progress information.

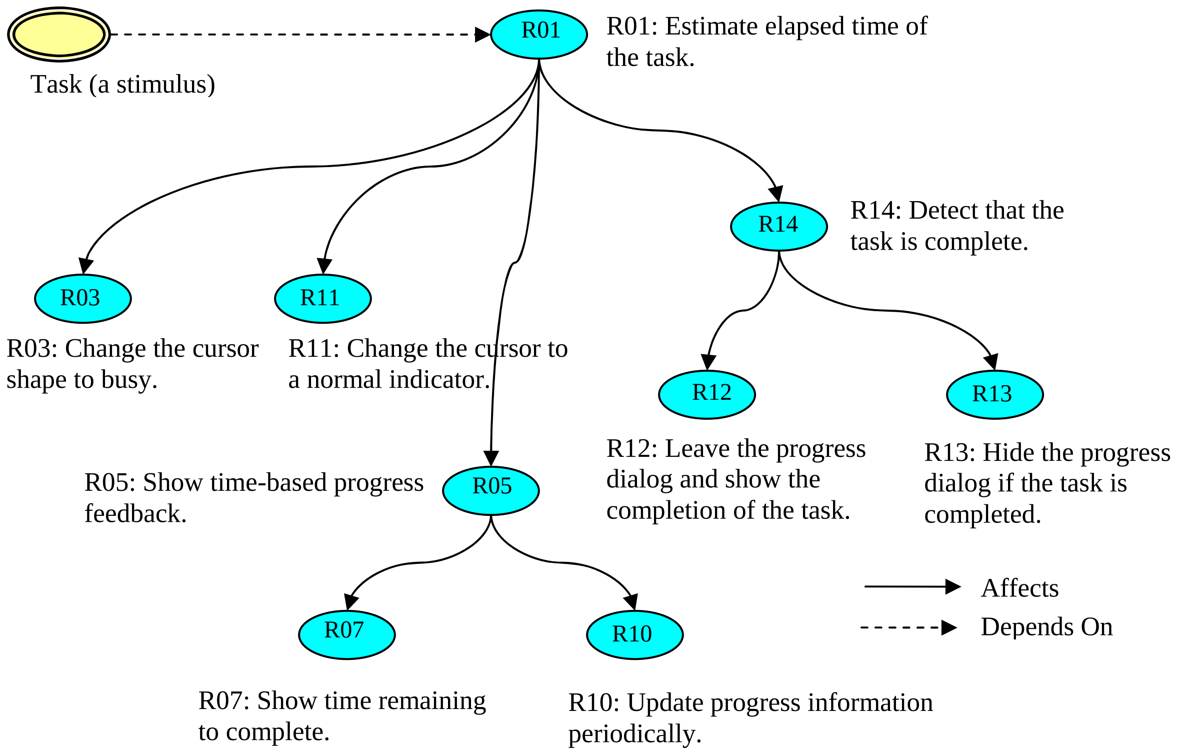

Task (a stimulus)
R01
R01: Estimate elapsed time of the task.
R14
R14: Detect that the task is complete.
R03
R03: Change the cursor shape to busy.
R11
R11: Change the cursor to a normal indicator.
R12
R12: Leave the progress dialog and show the completion of the task.
R13
R13: Hide the progress dialog if the task is completed.
R05
R05: Show time-based progress feedback.
R07
R07: Show time remaining to complete.
R10
R10: Update progress information periodically.
Affects
Depends On

Figure 5: Responsibility Graph After Adding Time-Based Progress Feedback Responsibilities

3. The elapsed time for the task will be greater than 10 seconds, and the remaining time cannot be estimated within an accuracy of 20%.

   In this case, the responsibility graph is modified to contain the responsibilities shown in Figure 6. The R10 responsibility (update progress information periodically) is accompanied by the generation of a performance scenario that has a stimulus of the update period of time and a response of updating progress information.

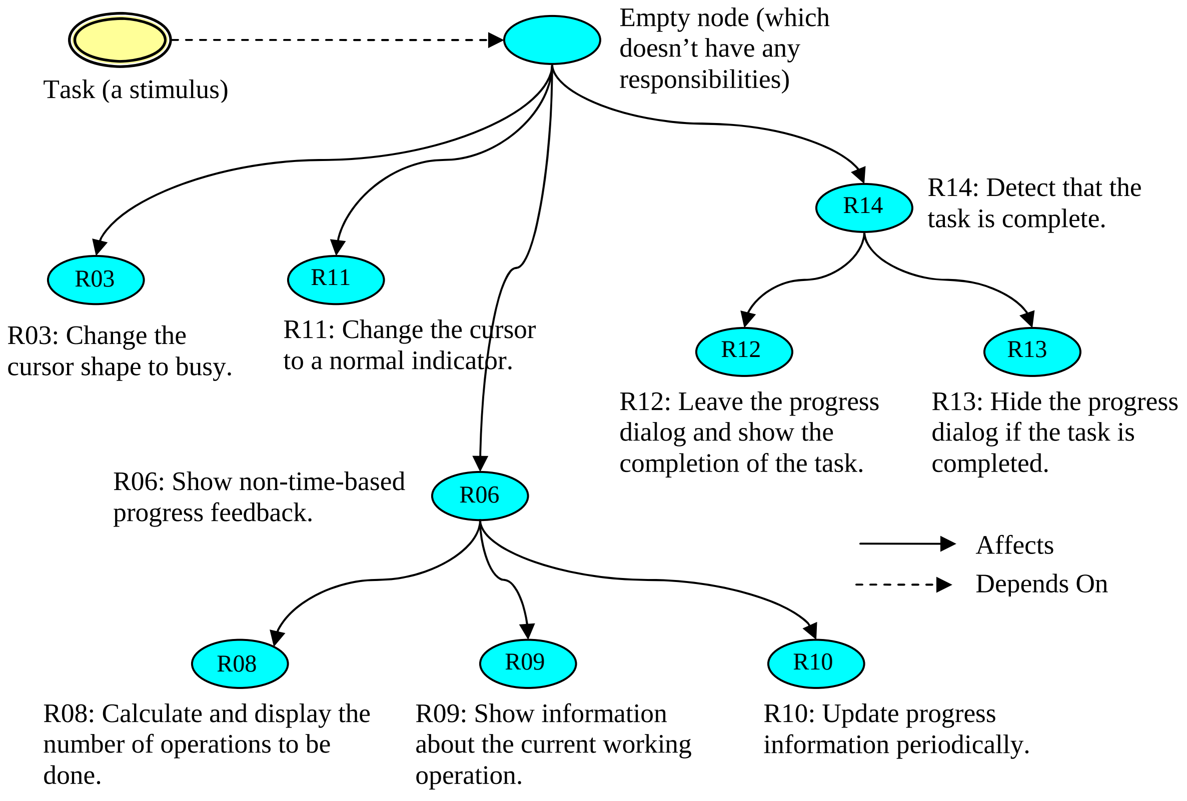

Task (a stimulus)
Empty node (which doesn't have any responsibilities)
R14
R14: Detect that the task is complete.
R03
R03: Change the cursor shape to busy.
R11
R11: Change the cursor to a normal indicator.
R12
R12: Leave the progress dialog and show the completion of the task.
R13
R13: Hide the progress dialog if the task is completed.
R06
R06: Show non-time-based progress feedback.
Affects
Depends On
R08
R08: Calculate and display the number of operations to be done.
R09
R09: Show information about the current working operation.
R10
R10: Update progress information periodically.

Figure 6: Responsibility Graph After Adding Non-Time-Based Progress Feedback Responsibilities

5.6 Summary

The complete ARL specification for the progress feedback USAP is given in Appendix A. It includes the preparation (determining what type of progress feedback should be shown) and the addition of the progress feedback responsibilities to the responsibility graph.

6 Cancellation Reasoning Framework

In this section, we describe the implementation of the cancellation reasoning framework. We follow the sequence described at the end of Section 4: focus on scenarios, organize responsibilities, derive an English description of the activities, and develop the code.

First, we present the general scenario for cancellation. Next, we describe how the ArchE UI is modified to enable the architect to specify the cancellation scenario. Finally, we describe the ARL implementation, including general responsibilities from the cancellation USAP and the implementation structure and description. The code for the UI portion and the reasoning framework are presented in Appendices C and D.

6.1 General Scenarios

The base scenario for cancellation is *The user initiates a long-running command, changes his/her mind, and wishes to terminate the command and restore the system to its state prior to invoking the command*.

A coupling exists between cancellation and progress feedback: because the ability to cancel should be provided for any command whose execution takes longer than two seconds, cancellation and progress feedback apply to the same set of commands. Typically, a cancel button is associated with various forms of progress feedback. We do not embed the coupling into the scenarios, but their trigger conditions are identical except for the case where the cancellation itself is a long-running task. In this case, there are two options:

1. The user is presented with progress feedback about the state of the cancel operation.
2. The progress feedback is embedded into the cancel operation either explicitly (by having it generate a progress scenario) or implicitly (by copying the progress feedback activities into the cancel operation).

Because there is no evidence that combining cancellation and progress feedback is more useful than separating them, we chose to do the latter.

The cancellation scenario allows for a partial implementation of the cancel operation and can use a response measure of the percentage of responsibilities satisfied by the design. The six-part scenario formulation for cancellation is shown in Table 4.

Table 4: Six-Part Scenario Formulation for Cancellation

Element	**Description**
Stimulus	Long-running task is initiated (e.g., cancel during file download from a Web site).
Stimulus Source	End user
Environment	At runtime
Artifact	System
Responses	• System provides the ability to cancel. • System shows progress feedback to the user while the system rolls back changes. • System provides feedback to the user that the cancellation is done.
Response Measure	Percentage of implemented responsibilities • Does the system implement above a certain percentage of total cancellation responsibilities?

6.2 Create the UI Configuration File for ArchE

ArchE derives the following information from a configuration file named *rfconfig.xml*:

- scenario type
- relationship type
- parameter type
- model elements

The *rfconfig.xml* file should be created using the ArchE Configurator tool. Each element in the general scenario can have specific types and units if the elements are used in the reasoning framework. For this cancellation reasoning framework, the elements may have limited types and units as shown in Table 5.

Table 5: Element Types and Units for Cancellation

Element	**Type**	**Default Type**	**Default Value**
Stimulus	Long-running task	Long-running task	N/A
Stimulus source	End user	End user	N/A
Environment	Runtime	Runtime	N/A
Artifact	System	System	N/A
Responses	Ability to cancel, progress feedback	Ability to cancel	N/A
Response measure	Boolean (yes/no) Percentage complete	Boolean (yes/no) Decimal	Yes 100%

If the new *rfconfig.xml* file is loaded by ArchE, the scenario wizard displays a screen similar to the one shown in Figure 2 on page 10.

For the complete *rfconfig.xml* file for this reasoning framework, see Appendix D.

6.3 Responsibilities

The cancellation USAP has 18 general responsibilities. If the command being cancelled can cancel itself, it performs many of the responsibilities. If it cannot cancel itself, the infrastructure must cancel it, and there must be some communication between the active command and the infrastructure. If the command has collaborating processes, they must be notified of the cancellation request.

The system should have the responsibilities shown in Table 6 to support a command being cancelled. The full cancellation USAP has been described by John [John 04].

Table 6: General Responsibilities for Cancellation

Responsibility	**Description**
R01	Expose a cancellation button or menu to the user.
R02	Listen for the cancel command or changes in the system environment.
R03	Save the initial state of the system.
R04	Show the user the response message that can prove the cancellation command is received immediately (within 150 milliseconds).
R05	Check if the active command can be cancelled directly at the time of cancellation.
R06	Cancel the active command directly.
R07	Ask the infrastructure to cancel the command.
R08	Check if the command has invoked any collaborating processes.
R09	Inform the collaborating processes of the invoking command’s cancellation.
R10	Check if the system is capable of rolling back all changes to the previous state.
R11	Restore the system state to its previous state right before the current operation.
R12	Restore the system state to its last saved state.
R13	Inform the user of any differences between the prior and restored states.
R14	Free all the resources the system used to run the cancelled command.
R15	Report to the user if the resource is not fully restored.
R16	Keep track of the resources that can be freed.
R17	Keep track of collaborating processes.
R18	Cancel the command using features of the infrastructure (e.g., Windows Task Manager).

6.4 Workflow Description

6.4.1 Background

When an architect specifies a cancellation type for a scenario, one of the scenario elements is the task for which cancellation must be provided. This task translates into a responsibility within ArchE and, subsequently, is assigned to various elements of the architecture.

The primary responsibility of the cancellation reasoning framework is to manipulate the responsibilities within ArchE. This manipulation consists of adding the necessary responsibilities (or ensuring that they are included). How the manipulated responsibilities are assigned to architectural elements is outside of the cancellation reasoning framework.

6.4.2 Identifying Unknowns

During preparation, any possible unknowns that affect the quality attribute model must be addressed. For cancellation, the possible unknowns are

1. Has a responsibility been linked to the task described in the scenario?

   If not, the architect must be prompted to make that linkage.

2. Is the infrastructure going to be involved in the cancellation?

   If the active command is blocked in some fashion, it may not be able to participate in its cancellation. In this case, the infrastructure will perform the cancellation. If the infrastructure has facilities to collaborate with the command being cancelled, they should be used. The architect is asked during preparation whether the infrastructure has such facilities.

3. Are the collaborating processes known to the designer, or does the system itself need to maintain a list?

   If the designer knows the collaborating processes, they can be hard-coded; if not, the system must detect the use of collaborating processes and maintain a list of them for notification.

6.4.3 Construct the Quality Attribute Model

The next step in a reasoning framework is to construct the quality attribute model. For the cancellation reasoning framework, the quality attribute model uses the answers to any possible unknowns to determine whether all responsibilities are already included. If they are, the evaluation says the scenario is satisfied. If they are not, the tactic section proposes—or in this case, performs—adding the necessary responsibilities to ArchE.

6.5 Implementation

This section elaborates the basic workflow described in Section 6.4. The responsibilities associated with cancellation are attached to the responsibility graph in the same fashion as progress feedback. That is, the scenario defines a task to be cancelled, that task must be linked to a responsibility (or responsibilities) in the responsibility graph, and the responsibilities that they are linked to become the parent responsibilities for any responsibility associated with the cancellation command.

6.5.1 Preparation

During preparation, the reasoning framework determines if it has the information necessary to proceed. Activities flow as follows:

1. Retrieve the task and the relevant responsibilities from the usability scenario.

   Several responsibilities of the target system might be affected by the task that is a stimulus of the scenario. The cancellation reasoning framework gets the responsibilities affected by the task defined in the usability scenario. If the scenario is not linked to any responsibilities, the architect is queried to determine this linkage.

2. Determine the assumed capabilities of the infrastructure.

   If a command cannot cancel itself because it is blocked on some resource or in an infinite loop, the infrastructure must act to cancel the command. Two problems could arise: (1) unless the infrastructure is informed of the resources being used by the cancelled command, it cannot free them and (2) the infrastructure must be informed of collaborating processes so that it can inform them in the event of a cancellation. If the infrastructure has such capabilities, the responsibilities of the command must include informing the infrastructure of any changes in resource utilization and collaborating processes.

3. Determine the extent of the designer's knowledge of collaborating processes.

   If the designer knows the identity of collaborating processes *a priori*, the list can be hard-coded. Otherwise, the system must collect this list during execution.

The information collected during preparation is made available so that tactics can be applied.

6.5.2 Instantiation and Evaluation

Instantiation refers to the creation of a quality attribute model from the architecture. The quality attribute model for cancellation has no parameters that depend on the elements of the architecture design. Evaluating the quality attribute model, therefore, consists of verifying that the required responsibilities are included in the responsibility graph. If they are, the scenario is satisfied; if they are not, they are added to the responsibility graph as described below.

6.5.3 Apply Tactics

In this section, we describe how the responsibility graph is modified to reflect the cancellation USAP. The general form was given in Figure 2 on page 10. The task considered in the scenario is linked to one or more responsibilities (parent responsibilities) in the responsibility graph. We identify three cases and distinguish them based on the architect's responses to the preparation questions. We assume that the designer (1) will know the capabilities of the infrastructure and (2) will *not* know the list of collaborating processes: the command must record them (R17: Keep track of collaborating processes). (Note that collaborating processes are not the processes inside the application: they are the processes that enable the application to execute the current command. For example, when downloading a file in a browser, the Web server process and an Internet connection program can be collaborating processes.)

1. The application can cancel the command.

   In this case, only application-level cancellation responsibilities are included, and the responsibility graph is modified to contain the responsibilities shown in Figure 7.

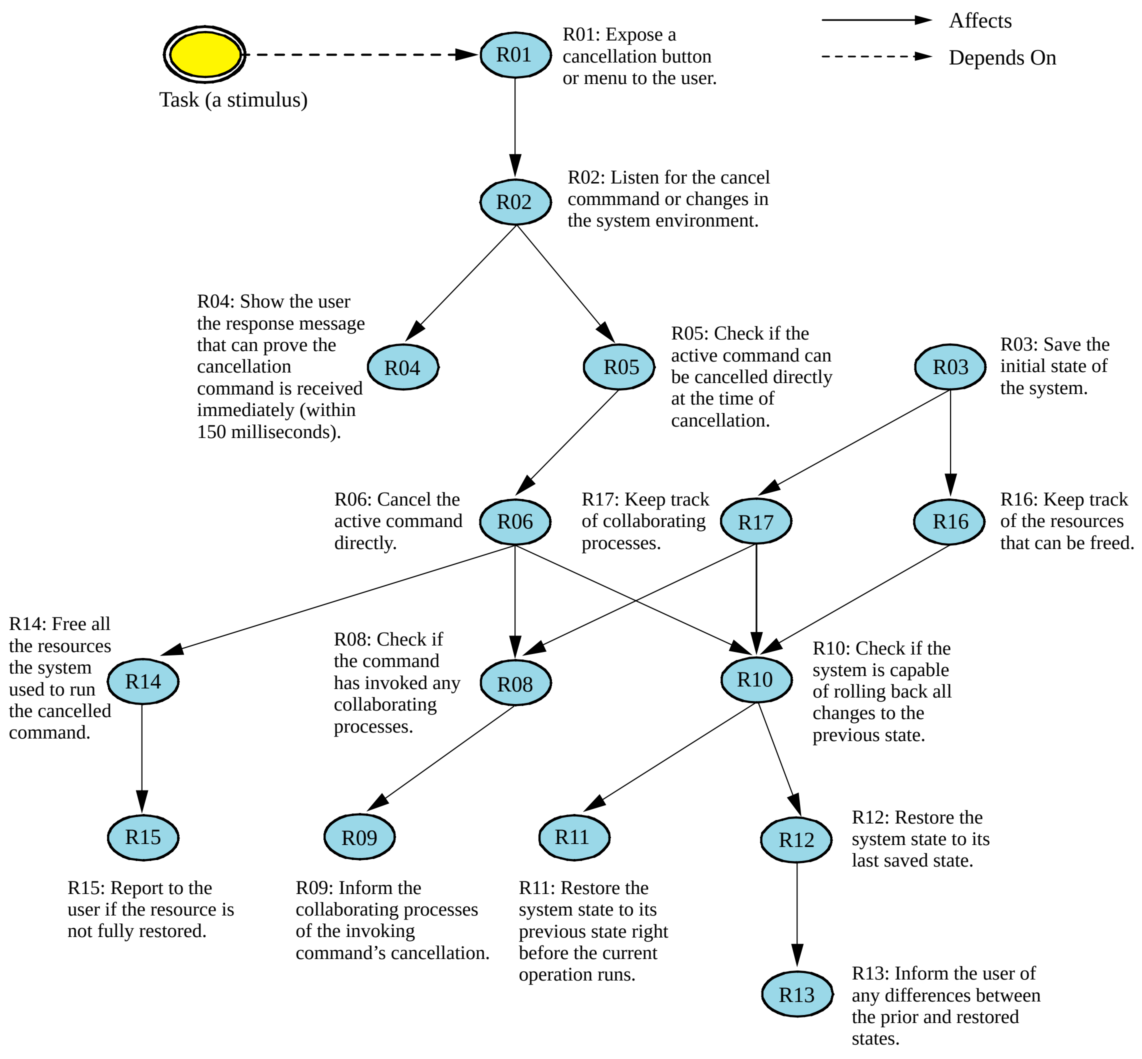

Affects
Depends On
Task (a stimulus)
R01
R01: Expose a cancellation button or menu to the user.
R02
R02: Listen for the cancel commmand or changes in the system environment.
R04
R04: Show the user the response message that can prove the cancellation command is received immediately (within 150 milliseconds).
R05
R05: Check if the active command can be cancelled directly at the time of cancellation.
R03
R03: Save the initial state of the system.
R06
R06: Cancel the active command directly.
R17
R17: Keep track of collaborating processes.
R16
R16: Keep track of the resources that can be freed.
R14
R14: Free all the resources the system used to run the cancelled command.
R08
R08: Check if the command has invoked any collaborating processes.
R10
R10: Check if the system is capable of rolling back all changes to the previous state.
R15
R09
R11
R12
R12: Restore the system state to its last saved state.
R15: Report to the user if the resource is not fully restored.
R09: Inform the collaborating processes of the invoking command’s cancellation.
R11: Restore the system state to its previous state right before the current operation runs.
R13
R13: Inform the user of any differences between the prior and restored states.

Figure 7: Responsibility Graph After Adding Application-Level Cancellation Responsibilities

2. The application cannot cancel the command directly and asks the infrastructure to cancel it by force.

   In this case, application- and infrastructure-level responsibilities must be included, and the responsibilities shown in Figure 8 are added as children to the task to which cancellation is being added.

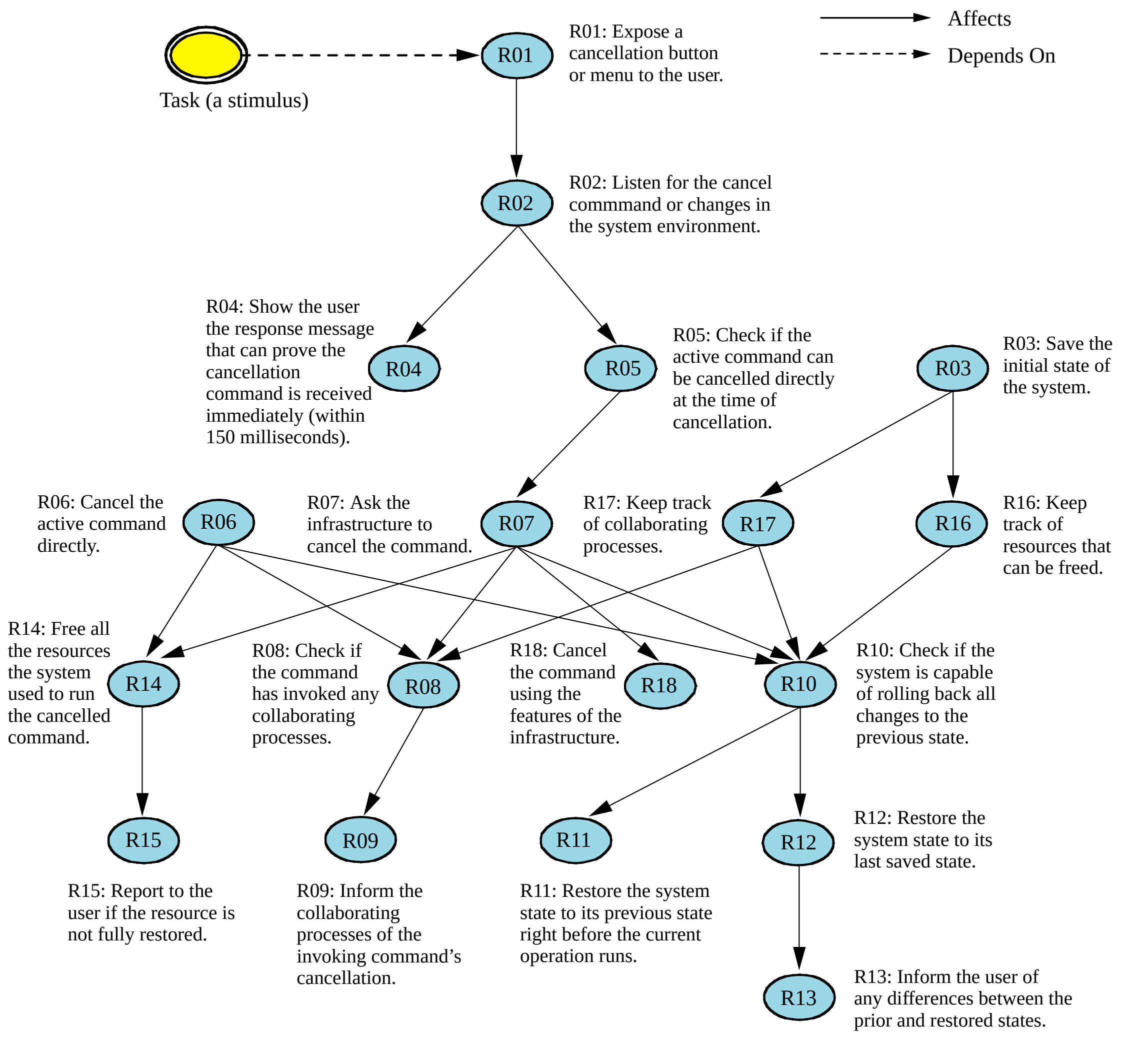
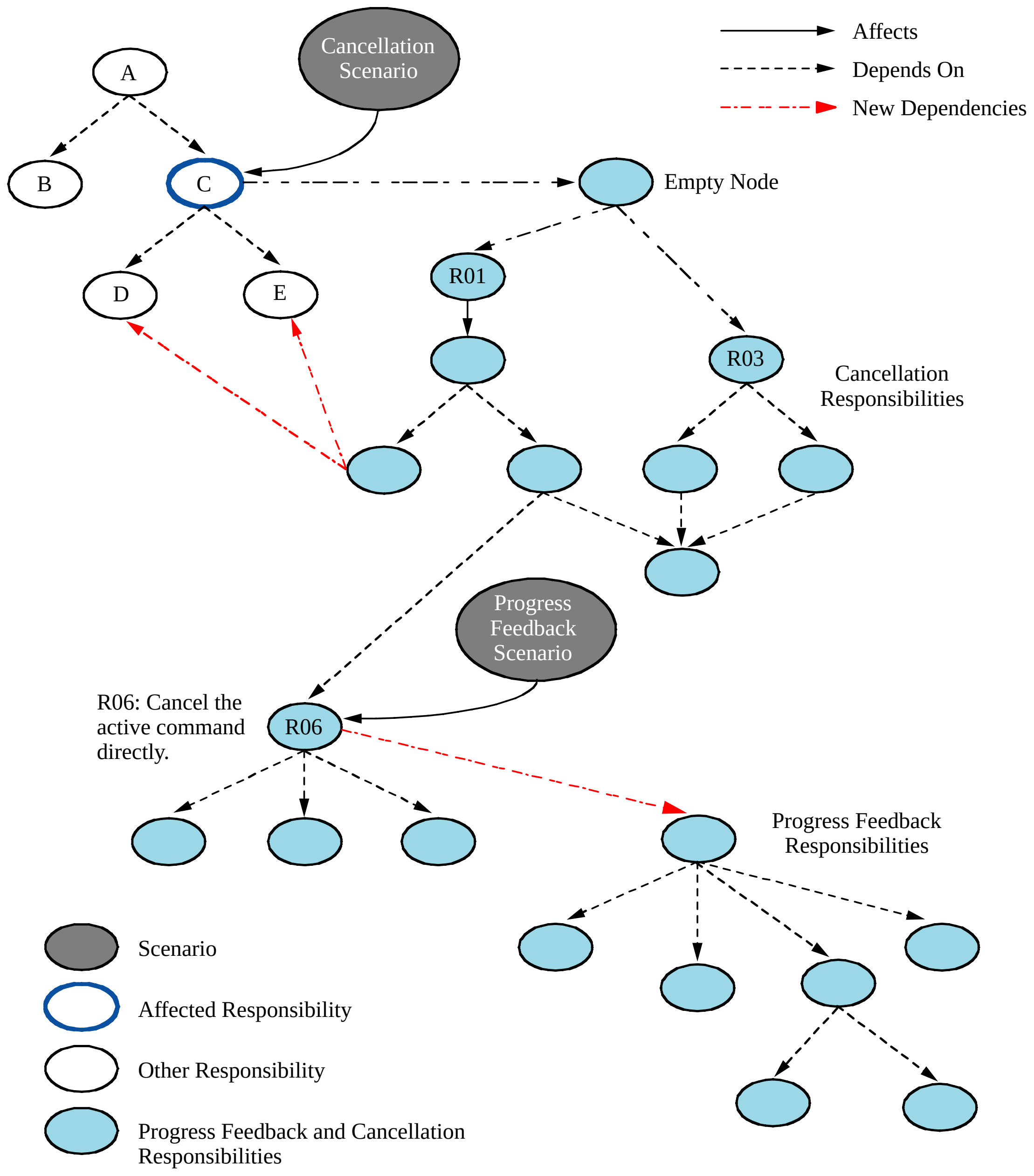

Affects
Depends On
Task (a stimulus)
R01
R01: Expose a cancellation button or menu to the user.
R02
R02: Listen for the cancel commmand or changes in the system environment.
R04: Show the user the response message that can prove the cancellation command is received immediately (within 150 milliseconds).
R04
R05
R05: Check if the active command can be cancelled directly at the time of cancellation.
R03
R03: Save the initial state of the system.
R06: Cancel the active command directly.
R06
R07: Ask the infrastructure to cancel the command.
R07
R17: Keep track of collaborating processes.
R17
R16
R16: Keep track of resources that can be freed.
R14: Free all the resources the system used to run the cancelled command.
R14
R08: Check if the command has invoked any collaborating processes.
R08
R18: Cancel the command using the features of the infrastructure.
R18
R10
R10: Check if the system is capable of rolling back all changes to the previous state.
R15
R09
R11
R12
R12: Restore the system state to its last saved state.
R15: Report to the user if the resource is not fully restored.
R09: Inform the collaborating processes of the invoking command's cancellation.
R11: Restore the system state to its previous state right before the current operation runs.
R13: Inform the user of any differences between the prior and restored states.
R13

Figure 8: Responsibility Graph After Adding Infrastructure-Level Cancellation Responsibilities

3. The cancellation itself is long running.

   In this case, the cancellation task requires progress feedback. A progress feedback scenario is generated, and the necessary responsibilities are added as shown in Figure 9.



Affects
Depends On
New Dependencies
Cancellation Scenario
A
B
C
D
E
Empty Node
R01
R03
Cancellation Responsibilities
Progress Feedback Scenario
R06: Cancel the active command directly.
R06
Progress Feedback Responsibilities
Scenario
Affected Responsibility
Other Responsibility
Progress Feedback and Cancellation Responsibilities

Figure 9: Responsibility Graph After Adding Progress Feedback Responsibilities

6.6 Summary of Cancellation

Adding a cancellation reasoning framework to ArchE is very much like adding the progress feedback reasoning framework: it mainly manipulates the responsibility graph of the system being designed. The exception is the creation of a progress feedback scenario if the cancellation itself is a long-running task.

The ARL code for cancellation is given in Appendices C and D.

7 Summary

In this technical note, we presented the incorporation of several aspects of a usability reasoning framework into ArchE using a specialized language, the ARL.

Incorporating a new reasoning framework involves specifying preparation, interpretation, evaluation, and tactics application steps. Once it is known how to specify these steps, the reasoning framework can be coded in the ARL.

The two reasoning frameworks described in this note do not employ the interpretation and evaluation steps. Therefore, we have emphasized manipulating the responsibility graph.

To complete the usability reasoning framework, additional scenarios would need to be implemented. We are driven in this work by the actual projects with which we interact, and completion of the usability reasoning framework will likely depend on finding a project for which it is useful.

Appendix A ARL Implementation for Progress Feedback

The progress feedback reasoning framework consists of four ARL files. Figure 10 shows the ARL namespaces that contain sets of rules that must execute in a predefined sequence. The ARL namespaces typically are also mapped one-to-one on Jess files with the extension *.clp*. The progress feedback reasoning framework does not need to suggest and try several tactics. Therefore, two namespaces—*SuggestProgressTactics* and *TryProgressTactics*—are empty.

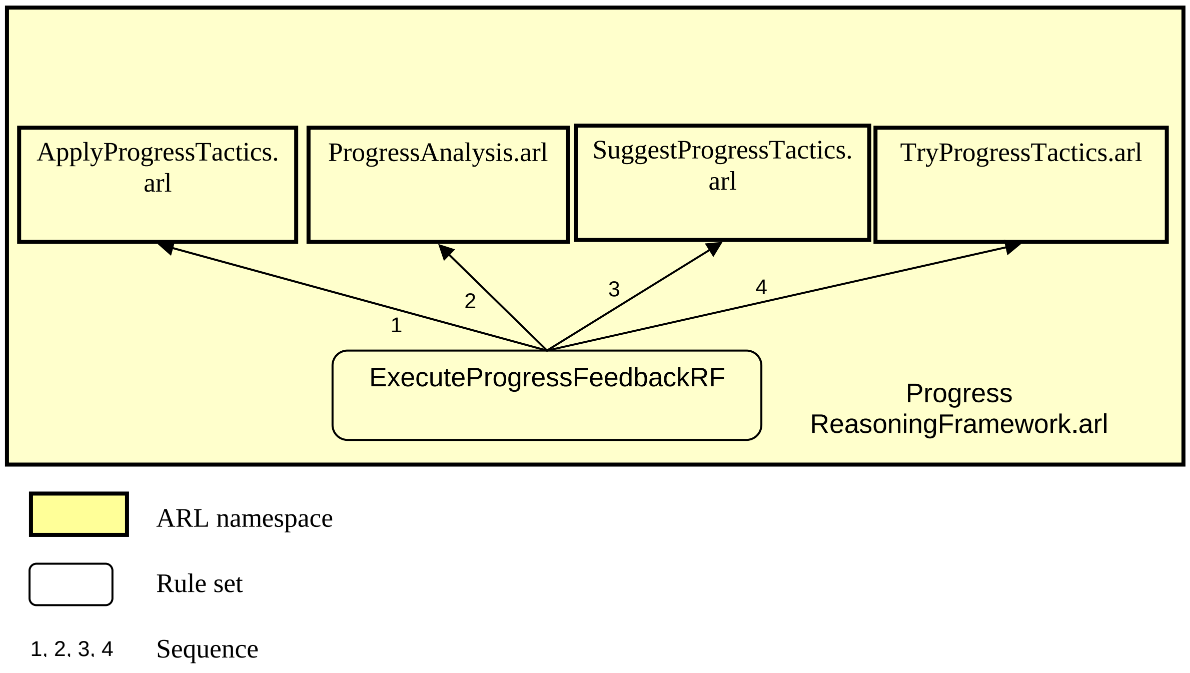


Figure 10: ARL Namespaces for Progress Feedback

ProgressReasoningFramework.arl

```
namespace ProgressReasoningFramework;

import MAIN;

/**
* Parameter: P_ExecutionTime
* This parameter defines the estimation of execution time for the long-running task
*/
```

```
type P_ExecutionTime {
  property int value;
  property String owner;      // Id of the owner of the property
  property String source; // Origin of the property value: ArchE, User
  property String status;     // Indicators for the property:nil, conflict
}

/**
* Parameter: P_Accuracy
* This parameter defines the accuracy of estimated time compared to previous
* esitmation
*/
type P_Accuracy {
  property int value;
  property String owner;      // Id of the owner of the property
  property String source;     // Origin of the property value: ArchE, User
  property String status;     // Indicators for the property:nil, conflict
}

/*
  Fact:  Keep track of what responsibilities are affected by a scenario
*/
type Node_affected {
  property MAIN::Scenarios scenario;
  property int responsibilityId;
  property int nodeId;
)

/*
  Fact: Kept track of what is the type of  assigned pattern for the long running task
*/
type FeedbackType{
  property String type;
}
/**
* The information about the tasks that users will initiate.
*/
type LongTask {
  property String name;               // name of the initiated task
  property MAIN::Scenarios scenario;  // parent scenario
  property int executiontime;         // estimated elapsetime to be complete
  property int priority;              // the priority of the task
  property int estimateaccuracy;      // the accuracy of the estimated time
  property int responsibilityId;
}

/*
* The information about the answer by users
* This fact will be asserted when users answer for the question that ArchE asked.
*/
type AnswerFromUser {
  property String questionId;
  property boolean answerAvailable;
  property List answer;
}

/*
* main function to execute this usability reasoning framework.
*/
rule ExecuteProgressRF{
  description: "rule to execute this reasoning framework";
  queries {
    exists (MAIN::Scenarios sc = getall MAIN::Scenarios where (quality ==
"Usability"));
  }
  actions {
  focus (ApplyProgressTactics, ProgressAnalysis, SuggestProgressTactics
TryProgressTactics);
  }
}
```

ApplyProgressTactics.arl

```
namespace ApplyProgressReasoningFramework;

import MAIN;
import ProgressReasoningFramework;

order ControlModelExecution {
  "initial" : DetermineAffectedNodes, CreateLongTaskFromScenario;
  "askquestion" : QEstimationAbilityOfSystem, QAccuracyCalculationAbilityOfSystem,
QGuaranteeAccuracyOfEstimation, QDetermineIfTheTaskCompletionShown;
  "applytactics" : ApplyWholeProgressFeedback, ApplyTaskBasedProgressFeedback,
ApplyTimeBasedProgressFeedback ;
}

rule DetermineAffectedNodes {
  description: "find which responsibilities are affected by the usability scenario";
  queries {
    MAIN::Scenarios sc = getall MAIN::Scenarios where quality == "Usability";
    MAIN::TranslationRelation tr = getall MAIN::TranslationRelation where parent ==
sc;
  }
  actions {
    new ProgressReasoningFramework::Node_affected ( scenario = sc;
        responsibilityId = tr.child;
        nodeId = getFactId(tr.child);
    );
  }
}

rule CreateLongTaskFromScenario{
  description: "create  all longtasks affected by usability scenario";
  queries {
    MAIN::Scenarios sc = getall MAIN::Scenarios
              where  quality == "usability"   && stimulusType == "longrunning";

    ProgressReasoningFramework::Node_affected nodes =
              getall ProgressReasoningFramework::Node_affected
          where scenario == sc ;
  }
  actions {
      new ProgressReasoningFramework::LongTask (
            name = sc.stimulusText;
            scenario = sc;
            responsibilityId = nodes.responsibilityID;
      );
    }
  }
}

// question rule for asking designers the estimation ability of system
question QEstimationAbilityOfSystem {
     description: "ask designer if the system would  have the estimation
     responsibility for long running task.";
  queries {
    logical (MAIN::Scenarios sc = getall MAIN::Scenarios where quality ==
“Usability”);
    Responsibilities rr = GetDependentResponsibilities(sc);
    logical (getall MAIN::TranslationRelation where parent == sc);
  }
  asking {
    MAIN::AskQuestion q = new MAIN::AskQuestion (
        questionId = "estimateElapsedTime";
        parent = sc;
        defaultAnswers = "yes";
        parameters = rr;
        affectedFacts = create$(sc, rr);
        log = nil;
    );
  }
  actions {
    // assert new fact named AnswerFromUser
    new ProgressReasoningFramework::AnswerFromUser(
        questionId = q.questionId;
        answerAvailable = q.answerAvailable;
        answer = q.answer;
        );

    // assign the type of progress feedback
    // if the system doesn't have estimation features, task based pattern will be
applied
    if (  q.answerAvailable == true && q.answer == "no" ) {
      new ProgressReasoningFramework::FeedbackType(
        type = "taskbased";
        );
    }
    q.log = true;
  }
}

// question rule for asking designers the estimation ability of system
question QAccuracyCalculationAbilityOfSystem {
  description: "ask designer if the system can calculate the accuracy of estimation or
if the estimation is always accurate or inaccurate. ";
  queries {
    logical (MAIN::Scenarios sc = getall MAIN::Scenarios where quality ==
"Usability");
    Responsibilities rr = GetDependentResponsibilities(sc);
    logical (getall MAIN::TranslationRelation where parent == sc);

    // check if the designer answered yes for the question - estimateElapsedTime
    exists (getall ProgressReasoningFramework::AnswerFromUser
    where questionId =="estimateElapsedTime" && answerAvailable == true && answer ==
"yes");
  }
  questions {
    MAIN::AskQuestion q = new MAIN::AskQuestion (
        questionId = "calculateAccuarcy";
        parent = sc;
        defaultAnswers = "yes";
        parameters = rr;
        affectedFacts = create$(sc, rr);
        log = nil;
    );
  }
  actions {
        // assert new fact named AnswerFromUser
    new ProgressReasoningFramework::AnswerFromUser(
        questionId = q.questionId;
        answerAvailable = q.answerAvailable;
        answers = q.answer;
        );
          // assign the type of progress feedback
          // if the system has both estimation and accuracy calculation features,
          // full  pattern will be applied
    if (  q.answerAvailable == true && q.answer == "yes" ) {
      new ProgressReasoningFramework::FeedbackType(
        type = "fullpattern";
        );
    }
    q.log = true;
  }
}

// question rule for asking designers the accuracy of the estimation is under 20% or
not
question QGuaranteeAccuracyOfEstimation{
  description: "ask designer the accuracy of estimation done by the system whether it
is guaranteed as accurate or not ";
  queries {
    logical (MAIN::Scenarios sc = getall MAIN::Scenarios where quality ==
"Usability");
    Responsibilities rr = GetDependentResponsibilities(sc);
    logical (getall MAIN::TranslationRelation where parent == sc);
    // check if the designer answered yes for the question - estimateElapsedTime
    exists (getall ProgressReasoningFramework::AnswerFromUser
    where questionId =="estimateElapsedTime" && answerAvailable == true && answer ==
"yes");
    // check if the desginer answered no for the question - calculateAccuarcy
    exists (getall ProgressReasoningFramework::AnswerFromUser
    where questionId =="calculateAccuarcy" && answerAvailable == true && answer ==
"no");
  }
  questions {
    MAIN::AskQuestion q = new MAIN::AskQuestion (
        questionId = "guaranteeAccuarcy";
        parent = sc;
        defaultAnswers = "yes";
        parameters = rr;
        affectedFacts = create$(sc, rr);
        log = nil;
    );
  }
  actions {
    // assert new fact named AnswerFromUser
    new ProgressReasoningFramework::AnswerFromUser(
        questionId = q.questionId;
        answerAvailable = q.answerAvailable;
        answers = q.answer;
        );

    // assign the type of progress feedback
    // if the system has  estimation but doesn't have  accuracy calculation features,
    // if the accuracy is guaranteed  under 20%, time-based  pattern will be applied
    if (  q.answerAvailable == true && q.answer == "yes" ) {
      new ProgressReasoningFramework::FeedbackType(
        type = "timebased";
        );
    }
    // if the accuracy is not guaranteed  under 20%, time-based  pattern will be
applied
    else if (  q.answerAvailable == true && q.answer == "yes" ) {
      new ProgressReasoningFramework::FeedbackType(
        type = "taskbased";
        );
    }
    q.log = true;
  }
}

// question rule for asking designers if the designer or end user
// will determine that the system show the task completion dialog
question QDetermineIfTheTaskCompletionShown {
  description: "ask designer who would determine if  system shows the task completion
dialog. ";
  queries {
    logical (MAIN::Scenarios sc = getall MAIN::Scenarios where quality ==
"Usability");
    Responsibilities rr = GetDependentResponsibilities(sc);
    logical (getall MAIN::TranslationRelation where parent == sc);
  }
  questions {
    MAIN::AskQuestion q = new MAIN::AskQuestion (
        questionId = "showTaskcompletion";
        parent = sc;
        defaultAnswers = "enduser";
        parameters = rr;
        affectedFacts = create$(sc, rr);
        log = nil;
    );
  }
  actions {
    // assert new fact named AnswerFromUser
    new ProgressReasoningFramework::AnswerFromUser(
        questionId = q.questionId;
        answerAvailable = q.answerAvailable;
        answers = q.answer;
        );
    q.log = true;
  }
}

rule ApplyWholeProgressFeedback{
  description: "Apply whole feedback pattern if the designer answers yes for the
estimating elapsed time and the calculation of accuracy";
  queries {
    exists (MAIN::AskQuestion q = getall MAIN::AskQuestion
        where questionId  == "estimateElapsedTime" &&  answer == "yes" );
    MAIN::Responsibilities r = getall MAIN::Responsibilities;
    test (r = q.parent);

    // get affected nodes
    ProgressReasoningFramework::Node_affected  nodes =
                   getall ProgressReasoningFramework::Node_affected
                      where scenario = sc;
  }
  actions{
    // create the whole progress feedback and get the topmost node from the feedback
graphs.
    MAIN::Responsibilities topmost = GenerateWholeProgressResponsibilities();

    /* for each affected responsibility,  create a relation between the affected
responsibility
    and the topmost node of the feedback graph */
    foreach (singlenode  in nodes) {
      // create relation
      MAIN::ResponsibilityToResponsibilityRelation rel =
                   new MAIN::ResponsibilityToResponsibilityRelation(
            source = "ArchE";
                             parent = singlenode.responsibilityId;
                                                child = topmost;
                                                                           );
    // assign affected responsibilities to parent responsibilties of pattern nodes
      topmost.parent = singlenode.responsibilityID;

    // create parameters and  assign owners of these two parameters to each affected
node
      new ProgressReasoningFramework::P_EstimatedElapsedTime (
        owner = singlenode;  value = 10; /* 10sec */  source = "System";
                   );
      new ProgressReasoningFramework::P_Accuracy(
                   owner = singlenode;       value = 10; // 10 percents
                   source = "System";
                   );
    }
  }
}

/*
* populate the full progress responsibilities graph
*/
function MAIN::Responsibilities  GenerateWholeProgressResponsibilities (){
  // create the progress pattern
  // create responsibilities 01
  String res01 = "Estimate elapsed time of the task";
  MAIN::Responsibilities r01 = new MAIN::Responsibilities(
          name = "progress";   description = res01;
          source = "ArchE";
                          );

  // create responsibilities 04
  String res04 = "Calculate the accuracy of the time estimate";
  MAIN::Responsibilities r04 = new MAIN::Responsibilities(
          name = res04;  description = res04; source = "ArchE";
          );
  // connect it to the parent responsibilities
```

```
MAIN::ResponsibilityToResponsibilityRelation rel0104 =
        new MAIN::ResponsibilityToResponsibilityRelation(
          parent = r01;  child = r04;       source = "ArchE";
        );
// create responsibilities 03
String res03 = "Change the cursor shape to busy";
MAIN::Responsibilities r03 = new MAIN::Responsibilities(
        name = res03;  description = res03;  source = "ArchE";
        );
// create responsibilities 11
String res11 = "Change the cursor shape to busy";
MAIN::Responsibilities r11 = new MAIN::Responsibilities(
        name = res11;  description = res11;  source = "ArchE";
        );

// connect it to the parent responsibilities

MAIN::ResponsibilityToResponsibilityRelation rel0403 =
    new MAIN::ResponsibilityToResponsibilityRelation(
        parent = r04;  child = r03;  source = "ArchE";
        );
// connect it to the parent responsibilities

MAIN::ResponsibilityToResponsibilityRelation rel0411 =
    new MAIN::ResponsibilityToResponsibilityRelation(
        parent = r04;  child = r11; source = "ArchE";
        );

// create responsibilities 02
String res02 = "Determine the type of the progress indicator";
MAIN::Responsibilities r02 = new MAIN::Responsibilities(
        name = res02;  description = res02; source = "ArchE";
        );

// connect it to the parent responsibilities

MAIN::ResponsibilityToResponsibilityRelation rel0402 =
    new MAIN::ResponsibilityToResponsibilityRelation(
        parent = r04;  child = r02; source = "ArchE";
        );

  // create responsibilities 05
String res05 = "Show time-based progress feedback";
MAIN::Responsibilities r05 = new MAIN::Responsibilities(
        name = res05;  description = res05; source = "ArchE";
        );

// create responsibilities 06
String res06 = "Show task-based progress feedback";
MAIN::Responsibilities r06 = new MAIN::Responsibilities(
        name = res06;  description = res06; source = "ArchE";
        );

// create responsibilities 14
String res14 = "Detect that the task is complete";
MAIN::Responsibilities r14 = new MAIN::Responsibilities(
        name = res14;  description = res14; source = "ArchE";
        );

// connect it to the parent responsibilities

MAIN::ResponsibilityToResponsibilityRelation rel0205 =
    new MAIN::ResponsibilityToResponsibilityRelation(
        parent = r02;  child = r05; source = "ArchE";
        );

  // connect it to the parent responsibilities

MAIN::ResponsibilityToResponsibilityRelation rel0206 =
    new MAIN::ResponsibilityToResponsibilityRelation(
        parent = r02;  child = r06; source = "ArchE";
        );
```

```
// connect it to the parent responsibilities

MAIN::ResponsibilityToResponsibilityRelation rel0214 =
    new MAIN::ResponsibilityToResponsibilityRelation(
        parent = r02;  child = r14;  source = "ArchE";
        );

// create responsibilities 12
String res12 = "Leave the progress dialog and show the completion of the task";
MAIN::Responsibilities r12 = new MAIN::Responsibilities(
        name = res12;  description = res12; source = "ArchE";
        );

// create responsibilities 13
String res13 = "Hide the progress dialog if the task is completed";
MAIN::Responsibilities r13 = new MAIN::Responsibilities(
        name = res13;  description = res13;        source = "ArchE";
        );

// connect it to the parent responsibilities

MAIN::ResponsibilityToResponsibilityRelation rel1412 =
            new MAIN::ResponsibilityToResponsibilityRelation(
        parent = r14;  child = r12;        source = "ArchE";
        );

// connect it to the parent responsibilities

MAIN::ResponsibilityToResponsibilityRelation rel1413 =
            new MAIN::ResponsibilityToResponsibilityRelation(
        parent = r14;  child = r13;        source = "ArchE";
        );

// create responsibilities 7
String res7 = "Show time remaining to complete";
MAIN::Responsibilities r7 = new MAIN::Responsibilities(
        name = res7; description = res7;   source = "ArchE";
        );

// create responsibilities 10
String res10 = "Update progress information periodically";
MAIN::Responsibilities r10 = new MAIN::Responsibilities(
        name = res10;  description = res10; source = "ArchE";
        );

// create responsibilities 08
String res08 = "Calculate and display the number of operations to be done";
MAIN::Responsibilities r08 = new MAIN::Responsibilities(
        name = res08;  description = res08; source = "ArchE";
        );

// create responsibilities 09
String res09 = "Show the information of current working operation";
MAIN::Responsibilities r09 = new MAIN::Responsibilities(
        name = res09;  description = res09; source = "ArchE";
        );

// connect it to the parent responsibilities

MAIN::ResponsibilityToResponsibilityRelation rel0507 =
            new MAIN::ResponsibilityToResponsibilityRelation(
        parent = r05;  child = r07; source = "ArchE";
        );

// connect it to the parent responsibilities

MAIN::ResponsibilityToResponsibilityRelation rel0510 =
            new MAIN::ResponsibilityToResponsibilityRelation(
                   parent = r05;    child = r10;
                   source = "ArchE";
                       );
```

```
  // connect it to the parent responsibilities

  MAIN::ResponsibilityToResponsibilityRelation rel0610 =
      new MAIN::ResponsibilityToResponsibilityRelation(
                     parent = r06;    child = r10;
                     source = "ArchE";
                         );

  // connect it to the parent responsibilities

  MAIN::ResponsibilityToResponsibilityRelation rel0608 =
      new MAIN::ResponsibilityToResponsibilityRelation(
                     parent = r06;    child = r08;
                     source = "ArchE";
                         );

  // connect it to the parent responsibilities

  MAIN::ResponsibilityToResponsibilityRelation rel0609 =
      new MAIN::ResponsibilityToResponsibilityRelation(
                     parent = r06;    child = r09;
                     source = "ArchE";
                         );
  return r01;
}

rule ApplyTaskBasedProgressFeedback {
  description: "Apply Taskbased feedback pattern if the designer answers
                  (1) no for the estimating elapsed time or
        (2) yes for the estimation,  no for calculating accuracy,
                  and the accuracy is worse than 20%,";
  queries {
    exists (ProgressReasoningFramework::AnswerFromUser an1 =
                  getall ProgressReasoningFramework::AnswerFromUser
        where questionId == "estimateElapsedTime");

    ProgressReasoningFramework::AnswerFromUser an2 =
                  getall ProgressReasoningFramework::AnswerFromUser
        where questionId == "calculateAccuarcy";
        answerAvailable = q.answerAvailable;
        answers = q.answer;

    (exists (MAIN::AskQuestion q = getall MAIN::AskQuestion
            where questionId  == "estimateElapsedTime" &&  answer == "no" ) ||
     exists(MAIN::AskQuestion q = getall MAIN::AskQuestion
            where questionId  == "estimateElapsedTime" &&  answer == "yes";
    MAIN::Responsibilities r = getall MAIN::Responsibilities;
    test (r = q.parent);

    // get affected nodes
    ProgressReasoningFramework::Node_affected  nodes =
                           getall ProgressReasoningFramework::Node_affected
            where scenario = sc;
  }
  actions{
    // create the whole progress feedback and get the topmost node from the feedback
graphs.
    MAIN::Responsibilities topmost = GenerateTaskBasedProgressResponsibilities();

    /* for each affected responsibility,
         create a relation between the affected responsibility
    and the topmost node of the feedback graph */
    foreach (singlenode  in nodes) {
      // create relation
      MAIN::ResponsibilityToResponsibilityRelation rel =
                  new MAIN::ResponsibilityToResponsibilityRelation(
                     source = "ArchE"
                             parent = singlenode.responsibilityId;
         child = topmost;
               );
  // assign affected responsibilities to parent responsibilties of pattern nodes
    topmost.parent = singlenode.responsibilityID;

  // create parameters and  assign owners of these two parameters to each affected
node
    new ProgressReasoningFramework::P_EstimatedElapsedTime (
        owner = singlenode;  value = 10; /* 10sec */  source = "System";
                    );
    }
  }
}

/*
* populate the task based  progress responsibilities graph
*/
function MAIN::Responsibilities  GenerateTaskBasedProgressResponsibilities (){
  ...
}

rule ApplyTimeBasedProgressFeedback {
  description: "";
  queries {
    exists (ProgressReasoningFramework::AnswerFromUser an1 =
                            getall ProgressReasoningFramework::AnswerFromUser
            where questionId == "estimateElapsedTime");

    ProgressReasoningFramework::AnswerFromUser an2 =
                            getall ProgressReasoningFramework::AnswerFromUser
            where questionId == "calculateAccuarcy";
    answerAvailable = q.answerAvailable;
    answers = q.answer;

    ( exists (MAIN::AskQuestion q = getall MAIN::AskQuestion
            where questionId  == "estimateElapsedTime" &&  answer == "yes" ) ||
     exists(MAIN::AskQuestion q = getall MAIN::AskQuestion
            where questionId  == "estimateElapsedTime" &&  answer == "yes";
    MAIN::Responsibilities r = getall MAIN::Responsibilities;
    test (r = q.parent);

    // get affected nodes
    ProgressReasoningFramework::Node_affected  nodes =
                            getall ProgressReasoningFramework::Node_affected
            where scenario = sc;
  }
  actions{
    // create the whole progress feedback and get the topmost node from the feedback
graphs.
    MAIN::Responsibilities topmost = GenerateTimeBasedProgressResponsibilities();

    /* for each affected responsibility,
         create a relation between the affected responsibility
    and the topmost node of the feedback graph */
    foreach (singlenode  in nodes) {
      // create relation
      MAIN::ResponsibilityToResponsibilityRelation rel =
              new MAIN::ResponsibilityToResponsibilityRelation(

  source = "ArchE";
          parent = singlenode.responsibilityId;
                child = topmost;
                    );

    // assign affected responsibilities to parent responsibilties of pattern nodes
    topmost.parent = singlenode.responsibilityID;

    // create parameters and  assign owners of these two parameters to each affected
node
    new ProgressReasoningFramework::P_EstimatedElapsedTime (
        owner = singlenode;  value = 10; /* 10sec */  source = "System";
                    );
    new ProgressReasoningFramework::P_Accuracy(
        owner = singlenode;  value = 10; // 10 percents
        source = "System";   );
    }
  }
}
/*
* populate the time based  progress responsibilities graph
*/
function MAIN::Responsibilities  GenerateTimeBasedProgressResponsibilities (){
  ...
}
```

ProgressAnalysis.arl

```
namespace ProgressAnalysis;

import MAIN;
import Planner;
import ProgressReasoningFramework;

order ControlModelExecution {
        "initial": RemoveUnansweredProgressQuestions, DeleteOldResults;
        "check": ProgressAnalysisCalculateScenario;
        "addperfsc": CreatePerfScenario;
}

rule RemoveUnansweredProgressQuestions {
        description: "Remove all unanswered questions related to progress feedback";
        queries {
                exists ( MAIN::AskQuestion q = getall MAIN::AskQuestion
                where ( questionId == "estimateElapsedTime" ||  questionId ==
"calculateAccuarcy"
                              || questionId =="showTaskcompletion" ||  questionId ==
"guaranteeAccuarcy" )
                                                              &&  answerAvailable ==
null);
        }
        actions {
                delete q;
    }
}

rule DeleteOldResults {
        description: "delete old analysis results facts from memory";
        queries {
                MAIN::P_AnalysisResult r = getall MAIN::P_AnalysisResult  where
                        quality == "Usability" && reasoningFramework ==
"ProgressFeedback";
        }
        actions {
                delete r;
        }
}

rule ProgressAnalysisCalculateScenario {
        description: "execute the analysis model to check the progress feedback
scenario is satisfied";
        queries {
                MAIN::Scenarios sc = getall MAIN::Scenarios
                   where quality =="Usability" && measureType == boolean;
                MAIN::TranslationRelation tr = getall MAIN::TranslationRelation
                   where parent == sc;
                ProgressReasoningFramework::Node_affected  affected =
                   getall ProgressReasoningFramework::Node_affected
                              where scenario == sc && responsibilityId == tr.child
                        && nodeId == etFactId(tr.child);

        }
actions {
                // check the scenario is satisfied
                boolean res = CheckProgressFeedback(affected);
                new MAIN::P_AnalysisResult (
                        owner = sc; value = res; source ="ArchE"; quality =
"Usability";
                        reasoningFramework = "ProgressFeedback";
                        isSatisfied = true;
                                                );
        }
}

function boolean CheckProgressFeedback(ProgressReasoningFramework::Node_affected af) [
        foreach ( singlenode in af ) {
                MAIN::Responsibilities res = getall MAIN::Responsibilities
                                where parent == af.responsibilityID &&  name ==
                             "Progress";
                if ( res.size <= 0 ) {
                        return false;
                }
        }
        return true;
}

rule CreatePerfScenario{
        description: "Create new performance scenario if there is no scenario related
to performance";
        queries {
                notexists ( MAIN::Scenarios sc = getall MAIN::Scenarios
                                                      where quality =="Performance"  &&
reasoningFramework == "RMA");
        }
        actions {
                new  Planner::C_AddScenario (
                        state = "final"; description = sc.description;  quality ==
"Performance" ;
                        stimulusText = ""; stimulusType ="";  stimulusUnit ="";
stimulusValue ="";
                        sourceText = ""; sourceType ="";  sourceUnit =""; sourceValue
="";
                        artifactText = ""; artifactType ="";  artifactUnit ="";
artifactValue ="";
                        environmentText = ""; environmentType ="";  environmentUnit
=""; environmentValue ="";
                        responseText = ""; responseType ="";  responseUnit ="";
responseValue ="";
                        measureText = ""; measureType ="";  measureUnit ="";
measureValue ="";
                        );
      }
}
```

SuggestProgressTactics.arl

Blank

TryProgressTactics.arl

Blank

Appendix B rfconfig.xml for Progress Feedback

This configuration file will be used in the ArchE application to create a new quality attribute scenario.

```
<?xml version="1.0" encoding="utf-8"?>
<rf id="progress" name="Usability RF v0.1" version="0.1">
    <scenarioTypes>
        <scenarioType desc="Tooltip/help for usability scenario type"
            name="Progress feedback pattern" tID="Usability">
                    <parts>
                <part defaultText="Type stimulus here"
                    defaultTypeId="Task"
                    defaultUnitId="" defaultValue="" partType="STIMULUS">
                    <types>
                        <type name="Task" tID="Task"/>
                    </types>
                    <units />
                </part>
                <part defaultText="Type source of stimulus here"
                    defaultTypeId="Enduser" defaultUnitId=""
                    defaultValue="" partType="SOURCE_OF_STIMULUS">
                    <types>
                        <type name="System" tID="System"/>
                        <type name="End user" tID="Enduser"/>
                     </types>
                    <units/>
                </part>
                <part defaultText="Type environment here"
                    defaultTypeId="Runtime" defaultUnitId=""
                    defaultValue="" partType="ENVIRONMENT">
                    <types>
                        <type name="Runtime" tID="Runtime"/>
                    </types>
                    <units/>
                </part>
                <part defaultText="Type artifact here"
                    defaultTypeId="System" defaultUnitId=""
                    defaultValue="" partType="ARTIFACT">
                    <types>
                        <type name="System" tID="System"/>
                    </types>
                    <units/>
                </part>
                <part defaultText="Type response here"
                    defaultTypeId="Responsibilities" defaultUnitId=""
                    defaultValue="" partType="RESPONSE">
                    <types>
                        <type name="Responsibilities" tID="Responsibilities"/>
                    </types>
                    <units/>
                </part>
                <part defaultText="Type response measure here"
                    defaultTypeId="ProgressIndicator"
                    defaultUnitId="boolean" defaultValue="yes"
                      partType="RESPONSE_MEASURE">
                    <types>
                        <type name="Progress Indicator" tID="ProgressIndicator"/>
                        <type name="Cursor Shape" tID="CursorShape"/>
                    </types>
                    <units>
                        <unit name="boolean" tID="boolean"/>
                    </units>
                </part>
            </parts>
        </scenarioType>
    </scenarioTypes>
    <relationshipTypes/>
    <parameterTypes>
     <parameterType dataType="double" defaultValue="10" desc="Tooltip/help for
     estimated exec time" name="Elapsed time (sec)"
     tID="ProgressFeedbackReasoningFrameworks::P_ElapsedExecutionTime"/>
     <parameterType dataType="double" defaultValue="15" desc="Tooltip/help for
     accuracy of estimated exec time" name="Percentage (%)"
     tID="ProgressFeedbackReasoningFrameworks::P_Accuracy"/>
    </parameterTypes>
    <responsibilityParameters>
               <parameterType tID="
ProgressFeedbackReasoningFrameworks::P_ElapsedExecutionTime"/>
     <parameterType tID=" ProgressFeedbackReasoningFrameworks::P_Accuracy"/>
    </responsibilityParameters>
<model />
</rf>
```

Appendix C ARL Implementation for Cancellation

The cancellation reasoning framework consists of five ARL files. Figure 11 shows the ARL namespaces that contain sets of rules that must execute in a predefined sequence. The ARL namespaces typically are also mapped one-to-one on Jess files with the extension *.clp*. The reasoning framework does not have to consider the initial design because it does not affect the current design and responsibilities of the system. In addition, it does not need to suggest and try several tactics. Therefore, three namespaces—*InitialCancelDesign*, *SuggestCancelTactics*, and *TryCancelTactics*—are empty.

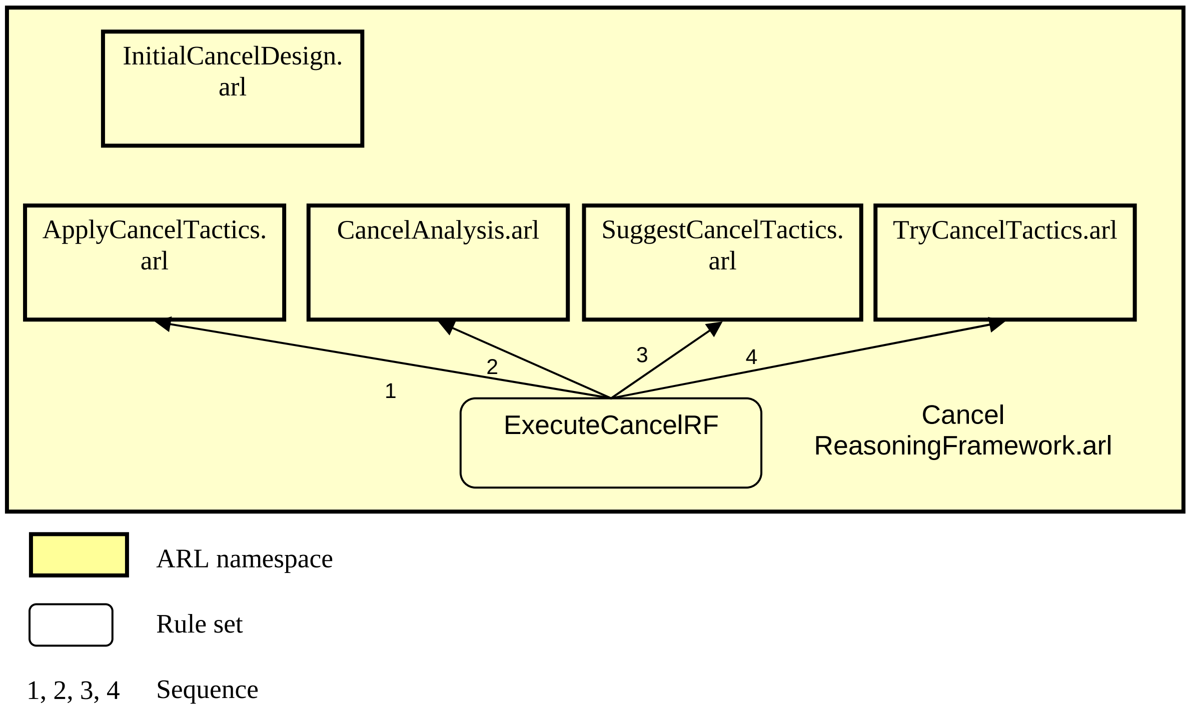


Figure 11: ARL Namespaces for Cancellation

CancelReasoningFramework.arl

```
type Node_affected {
        MAIN::Scenarios scenario;
        String responsibilityId;
        String nodeId;
}

type P_LevelOfCancel {
        String value;
}
```

```
type P_AbilityToTraceCollaboratingProcesses {
       String value;
}

type P_ListOfCollaboratingProcesses {
       List responsibilties;
}

/**
* The information about the tasks that users will initiate.
*/
type LongTask {
       String name;                  // name of the initiated task
       MAIN::Scenarios scenario;     // parent scenario
       int executiontime;            // estimated elapsetime to be complete
       int priority;                 // the priority of the task
       int estimateaccuracy;         // the accuracy of the estimated time
       String responsibilityId;
}

/*
* The information about the answer by users
* This fact will be asserted when users answer for the question that ArchE asked.
*/
type AnswerFromUser {
       String questionId;
       boolean answerAvailable;
       List answer;
}

/*
* main function to execute this usability reasoning framework.
*/
rule ExecuteCancelRF{
       comment: "rule to execute this reasoning framework";
       queries {
              exists (getall MAIN::Scenarios where (quality == "Cancellation"));
       }
       actions {
              focus ("ApplyCancelTactics", "CancelAnalysis", "SuggestCancelTactics",
                 "TryCancelTactics");
       }
}
```

ApplyCancelTactics.arl

```
rule DetermineAffectedNodes {
       comment: "find which responsibilities are affected by the usability scenario";
       queries {
              MAIN::Scenarios sc = getall MAIN::Scenarios where (quality ==
"Cancellation");
              MAIN::TranslationRelation tr = getall MAIN::TranslationRelation
         where (parent == sc);
       }
       actions {
              new CancelReasoningFramework::Node_affected ( scenario = sc,
                             responsibilityId = tr.child,
                             nodeId = tr.child);
       }
}
```

```
rule CreateTaskFromScenario{
        comment: "create all task to be cancelled ";
        queries {
                MAIN::Scenarios sc = getall MAIN::Scenarios
                                where(quality == "Cancellation")&& (stimulusType ==
"longrunning");

                CancelReasoningFramework::Node_affected nodes =
              getall CancelReasoningFramework::Node_affected where (scenario == sc);
        }
        actions {
                new CancelReasoningFramework::LongTask (
                        name = sc.stimulusText,
                        scenario = sc,
                        responsibilityId = nodes.responsibilityId);
        }
}

// question rule for asking designers the estimation ability of system
rule AskLevelOfCancel{
        comment: "ask designer if the system would have only application level
cancellation.";
        queries {
                logical (MAIN::Scenarios sc = getall MAIN::Scenarios
                   where (quality == "Cancellation"));
                MAIN::TranslationRelation tr = getall MAIN::TranslationRelation
                   where (parent == sc);
                logical (getall MAIN::TranslationRelation where (parent == tr.child));
        }
        actions {
                MAIN::AskQuestion q = new MAIN::AskQuestion (
                        questionId = "LevelOfCancellation",
                        parent = sc,
                        default = "Application",
                        parameters = tr.child,
                        log = null);
        }
}

rule getAnswerOfLevelFromUser {
        comment: "handle answer from user";
        queries {

                MAIN::AskQuestion q = getall MAIN::AskQuestion
                   where ( answerAvailable == true ) && ( log == null);
        }
        actions {
                // assert new fact named AnswerFromUser
                new CancelReasoningFramework::AnswerFromUser(
                        questionId = q.questionId,
                        answerAvailable = q.answerAvailable,
                        answer = q.answer
                        );

                // assign the type of progress feedback
                // if the system doesn't have estimation features,
                // task based pattern will be applied
                if ( q.answerAvailable == true ) {
                        new CancelReasoningFramework::P_LevelOfCancel(
                                value = q.answer);
                }
                q.log = true;
        }
}
```

```
rule AskAbilityToTraceCollaboratingProcess{
        comment: "ask designer about the collaborating system.";
        queries{
                logical(MAIN::Scenarios sc = getall MAIN::Scenarios
                   where (quality == "Cancellation"));
                MAIN::TranslationRelation tr = getall MAIN::TranslationRelation
                   where (parent == sc);
                logical(getall MAIN::TranslationRelation where (parent == tr.child));
        }
        actions{
                MAIN::AskQuestion q = new MAIN::AskQuestion (
                                questionId = "AbilityOfTracingCollaboration",
                                parent = sc,
                                default = "yes",parameters = tr.child, log = null);
        }
}

rule getAnswerOfAbilityToTraceCollaboration {
        comment: "handle answer from user";
        queries {
                MAIN::AskQuestion q = getall MAIN::AskQuestion
                   where ( answerAvailable == true ) && ( log == null);
        }
        actions {
                // assert new fact named AnswerFromUser
                new CancelReasoningFramework::AnswerFromUser(
                                questionId = q.questionId,
                                answerAvailable = q.answerAvailable,
                                answer = q.answer);

                // assign the type of progress feedback
                // if the system doesn't have estimation features,
                // task based pattern will be applied
                if (  q.answerAvailable == true ) {
                        new
CancelReasoningFramework::P_AbilityToTraceCollaboratingProcesses(
                                value = q.answer);
                }
                q.log = true;
        }
}

rule AskListOfollaboratingProcess{
        comment: "ask designer about the list of collaborating processes.";
        queries {
                logical (MAIN::Scenarios sc = getall MAIN::Scenarios
                        where (quality == "Cancellation"));
                MAIN::TranslationRelation tr = getall MAIN::TranslationRelation
                        where (parent == sc);
                logical (getall MAIN::TranslationRelation where (parent == tr.child));
                exists(getall
CancelReasoningFramework::P_AbilityToTraceCollaboratingProcesses
                where (value == "no" ));
        }
        actions {
                MAIN::AskQuestion q = new MAIN::AskQuestion (
                                questionId = "ListOfCollaborationProcesses",
                                parent = sc,
                                default = null,
                                parameters = tr.child,
                                log = null);
        }
}
```

```
rule getAnswerOfListOfCollaborationProcesses {
        comment: "handle answer from user";
        queries{
                MAIN::AskQuestion q = getall MAIN::AskQuestion
                   where ( answerAvailable == true ) && ( log == null);
        }
        actions{
                // assert new fact named AnswerFromUser
                new CancelReasoningFramework::AnswerFromUser(
                        questionId = q.questionId,
                        answerAvailable = q.answerAvailable,
                        answer = q.answer);

                // assign the type of progress feedback
                // if the system doesn't have estimation features,
                // task based pattern will be applied
                if (  q.answerAvailable == true ) {
                        new CancelReasoningFramework::P_ListOfCollaboratingProcesses(
                                responsibilties = q.answer);
                }
                q.log = true;
        }
}

rule ApplyApplicationCancelWithAbilityTracing{
        comment: "Apply application-level pattern if the designer answered
application";
        queries{

                exists(getall CancelReasoningFramework::P_LevelOfCancel
                   where (value == "Application"));
                MAIN::AskQuestion q = getall MAIN::AskQuestion
                                where (questionId  == "LevelOfCancellation") &&
                        (answer == "Application" );
                MAIN::Responsibilities r = getall MAIN::Responsibilities;
                test (r == q.parent);

                exists(getall
CancelReasoningFramework::P_AbilityToTraceCollaboratingProcesses
                   where (value =="yes"));
                // get affected nodes
                CancelReasoningFramework::Node_affected nodes =
                   getall CancelReasoningFramework::Node_affected;
        }
        actions{
                // create the cancellation pattern and get the topmost node from the
graphs.
                MAIN::Responsibilities topmost =
GenerateApplicationCancelWithCollaboration();

                /* for each affected responsibility,
                create a relation between the affected responsibility
                and the topmost node of the feedback graph */

                // create relation
                MAIN::ResponsibilityToResponsibilityRelation rel =
               new MAIN::ResponsibilityToResponsibilityRelation
                                (source = "ArchE", parent = nodes.responsibilityId,
        child = topmost       );
        }
}
```

```
rule ApplyApplicationCancelWithoutAbilityTracing{
        comment: "Apply application-level pattern if the designer answered
application";
        queries {
                exists(getall CancelReasoningFramework::P_LevelOfCancel
                   where (value == "Application"));
                MAIN::AskQuestion q = getall MAIN::AskQuestion
                               where (questionId  == "LevelOfCancellation")
                       &&  (answer == "Application" );
                MAIN::Responsibilities r = getall MAIN::Responsibilities;
                test (r == q.parent);

                exists(getall
CancelReasoningFramework::P_AbilityToTraceCollaboratingProcesses
                   where(value =="no"));
                // get affected nodes
                CancelReasoningFramework::Node_affected nodes =
                   getall CancelReasoningFramework::Node_affected;
        }
        actions{
                // create the cancellation pattern and get the topmost node from the
graphs.
                MAIN::Responsibilities topmost =
                   GenerateApplicationCancelWithoutCollaboration();

                // create relation
                MAIN::ResponsibilityToResponsibilityRelation rel = new
MAIN::ResponsibilityToResponsibilityRelation
                (source = "ArchE", parent = nodes.responsibilityId, child = topmost );
        }
}


rule ApplyInfrastructureCancelWithAbilityTracing{
        comment: "Apply system-level pattern if the designer answered application";
        queries{
                exists(getall CancelReasoningFramework::P_LevelOfCancel
                   where (value <> "Application"));
                MAIN::AskQuestion q = getall MAIN::AskQuestion
                               where (questionId  == "LevelOfCancellation")
                       && (answer <> "Application" );
                MAIN::Responsibilities r = getall MAIN::Responsibilities;
                test (r == q.parent);

                exists(getall
CancelReasoningFramework::P_AbilityToTraceCollaboratingProcesses
                   where (value =="no"));
                // get affected nodes
                CancelReasoningFramework::Node_affected nodes =
                   getall CancelReasoningFramework::Node_affected;
        }
        actions{
                // create the cancellation pattern and get the topmost node from the
graphs.
                MAIN::Responsibilities topmost =
                   GenerateInfrastructureCancelWithCollaboration();

                /* for each affected responsibility,
                create a relation between the affected responsibility
                and the topmost node of the graph */
                // create relation
                MAIN::ResponsibilityToResponsibilityRelation rel =
                   new MAIN::ResponsibilityToResponsibilityRelation
                               (source = "ArchE", parent = nodes.responsibilityId,
        child = topmost);
        }
}
```

```
rule ApplyInfrastructureCancelWithoutAbilityTracing{
        comment: "Apply system-level pattern if the designer answered application";
        queries{
                exists(getall CancelReasoningFramework::P_LevelOfCancel
                   where (value <> "Application"));
                MAIN::AskQuestion q = getall MAIN::AskQuestion
                               where (questionId  == "LevelOfCancellation")
                   && (answer <> "Application" );
                MAIN::Responsibilities r = getall MAIN::Responsibilities;
                test (r == q.parent);

                exists(getall
CancelReasoningFramework::P_AbilityToTraceCollaboratingProcesses
                   where (value =="no"));
                // get affected nodes
                CancelReasoningFramework::Node_affected nodes =
                   getall CancelReasoningFramework::Node_affected;
        }
        actions{
                // create the cancellation pattern and get the topmost node from the
graphs.
                MAIN::Responsibilities topmost =
              GenerateInfrastructureCancelWithoutCollaboration();

                // create relation
                MAIN::ResponsibilityToResponsibilityRelation rel =
              new MAIN::ResponsibilityToResponsibilityRelation
                        (source = "ArchE", parent = nodes.responsibilityId,child =
topmost);
        }
}

function MAIN::Responsibilities GenerateApplicationCancelWithCollaboration(){
        // create responsibilities 01
        String res01 = "Expose a button or menu to cancel the current operation to
user";
        MAIN::Responsibilities r01 = new MAIN::Responsibilities(
                name = "Cancellation", description = res01, source = "ArchE");

        String res02 = "Listen for the cancel command in the system environment ";
        MAIN::Responsibilities r02 = new MAIN::Responsibilities(
                name = res02,  description = res02, source = "ArchE");


        // connect it to the parent responsibilities

        MAIN::ResponsibilityToResponsibilityRelation rel0102 =
                                      new
MAIN::ResponsibilityToResponsibilityRelation(
                parent = r01,  child = r02,   source = "ArchE");

        String res04 = "Show the user the response message that can prove the
cancellation command is received immediately";
        MAIN::Responsibilities r04 = new MAIN::Responsibilities(
                name = res04,  description = res04, source = "ArchE");


        // connect it to the parent responsibilities

        MAIN::ResponsibilityToResponsibilityRelation rel0204 = new
        MAIN::ResponsibilityToResponsibilityRelation(
                parent = r02, child = r04,source = "ArchE");

        String res05 = "Check if the active command can be cancelled directly at the
time of cancellation";
        MAIN::Responsibilities r05 = new MAIN::Responsibilities(
                name = res05,  description = res05, source = "ArchE");
        // connect it to the parent responsibilities

        MAIN::ResponsibilityToResponsibilityRelation rel0205 =
                                        new
MAIN::ResponsibilityToResponsibilityRelation(
                parent = r02, child = r05,source = "ArchE");

        return r01;

}

order ControlModelExecution {
        initial : DetermineAffectedNodes, CreateTaskFromScenario;
        askquestions : AskLevelOfCancel, getAnswerOfLevelFromUser,
                                AskAbilityToTraceCollaboratingProcess,
                                getAnswerOfAbilityToTraceCollaboration,
                                AskListOfollaboratingProcess,
                                getAnswerOfListOfCollaborationProcesses      ;
        applytactics : ApplyApplicationCancelWithAbilityTracing,
                                ApplyApplicationCancelWithoutAbilityTracing,
                                ApplyInfrastructureCancelWithAbilityTracing,
                                ApplyInfrastructureCancelWithoutAbilityTracing;
}
```

CancelAnalysis.arl

```
rule RemoveUnansweredQuestions {
        comment: "Remove all unanswered questions related to cancellation";
        queries {
                MAIN::AskQuestion q = getall MAIN::AskQuestion
                                where ( questionId == "LevelOfCancellation" ) &&
                                (questionId == "ListOfCollaborationProcesses")
                                && (questionId =="AbilityOfTracingCollaboration")
                                && (answerAvailable == null);
        }
        actions {
                delete q;
    }
}

rule RemoveAnalysisResults {
        comment: "delete old analysis results facts from memory";
        queries{
                MAIN::P_AnalysisResult r = getall MAIN::P_AnalysisResult where
                        (quality == "Cancellation") && (reasoningFramework ==
"Cancellation");
        }
        actions{
                delete r;
        }
}

rule CheckNodeHasFeatures {
        comment: "execute the analysis model to check the scenario is satisfied";
        queries{
                MAIN::Scenarios sc = getall MAIN::Scenarios
                        where (quality =="Cancellation") && (measureType == "boolean");
                MAIN::TranslationRelation tr = getall MAIN::TranslationRelation
                        where (parent == sc);
                CancelReasoningFramework::Node_affected affected =
                        getall CancelReasoningFramework::Node_affected
                        where (scenario == sc) && (responsibilityId == tr.child)
                        && (nodeId == tr.child);

        }
        actions{
                // check the scenario is satisfied
                boolean res = CheckCancellationPattern(affected);
                new MAIN::P_AnalysisResult (
                        owner = sc, value = res, source ="ArchE", quality =
                        "Cancellation", reasoningFramework = "Cancellation",
                        isSatisfied = true);
        }
}

function boolean CheckCancellationPattern(CancelReasoningFramework::Node_affected af){
        boolean bReturn = true;
        MAIN::Responsibilities res = af.responsibilityId;
        if ( res.length <= 0 ) {
                bReturn = false;
        }
        return bReturn;
}

rule CreateProgressScenario{
        comment: "Create new scenario if there is no scenario related to progress";
        queries{
                notexists ( getall MAIN::Scenarios
                        where (quality =="ProgressFeedback"));
        }
        actions{
                new Planner::C_AddScenario (
                        state = "final", description = "",  quality = "Usability" ,
                        stimulusText = "", stimulusType ="",
              stimulusUnit ="", stimulusValue ="",
                        sourceText = "", sourceType ="",
              sourceUnit ="", sourceValue ="",
                        artifactText = "", artifactType ="",
              artifactUnit ="", artifactValue ="",
                        environmentText = "", environmentType ="",
              environmentUnit ="", environmentValue ="",
                        responseText = "", responseType ="",
              responseUnit ="", responseValue ="",
                        measureText = "", measureType ="",
              measureUnit ="", measureValue =""
                        );
     }
}

order ControlModelExecution {
        initial: RemoveUnansweredQuestions, RemoveAnalysisResults;
        check: CheckNodeHasFeatures;
        addnewsc: CreateProgressScenario;
}
```

SuggestCancelTactics.arl

Blank

TryCancelTactics.arl

Blank

InitialCancelDesign.arl

Blank

Appendix D   rfconfig.xml for Cancellation

This configuration file will be used in the ArchE application to create a new quality attribute scenario.

```
<?xml version="1.0" encoding="utf-8"?>
<rf id="cancellation" name="Cancellation Usability RF v0.1" version="0.1">
    <scenarioTypes>
        <scenarioType desc="Tooltip/help for cancellation usability scenario type"
            name="Canceling a command pattern" tID="Cancel-Usability">
            <parts>
                <part defaultText="Type stimulus here"
                    defaultTypeId="Task"
                    defaultUnitId="" defaultValue="" partType="STIMULUS">
                    <types>
                        <type name="Task" tID="Task"/>
                    </types>
                    <units />
                </part>
                <part defaultText="Type source of stimulus here"
                    defaultTypeId="Enduser" defaultUnitId=""
                    defaultValue="" partType="SOURCE_OF_STIMULUS">
                    <types>
                        <type name="System" tID="System"/>
                        <type name="End user" tID="Enduser"/>
                     </types>
                    <units/>
                </part>
                <part defaultText="Type environment here"
                    defaultTypeId="Runtime" defaultUnitId=""
                    defaultValue="" partType="ENVIRONMENT">
                    <types>
                        <type name="Runtime" tID="Runtime"/>
                    </types>
                    <units/>
                </part>
                <part defaultText="Type artifact here"
                    defaultTypeId="System" defaultUnitId=""
                    defaultValue="" partType="ARTIFACT">
                    <types>
                        <type name="System" tID="System"/>
                    </types>
                    <units/>
                </part>
                <part defaultText="Type response here"
                    defaultTypeId="AbilitytoCancel" defaultUnitId=""
                    defaultValue="" partType="RESPONSE">
                    <types>
                        <type name="Ability to cancel" tID="AbilitytoCancel "/>
                         <type name="Progress feedback" tID="ProgressFeedback"/>
                    </types>
                    <units/>
                </part>
                <part defaultText="Type response measure here"
                    defaultTypeId=" CommandCancelled"
                    defaultUnitId="boolean" defaultValue="yes"
                      partType="RESPONSE_MEASURE">
                    <types>
                        <type name="Command is cancelled" tID="CommandCancelled"/>
                        <type name="Showing progress indicator"
                      tID="ProgressIndicator"/>
                        <type name="Showing cursor shapes" tID="CursorShape"/>
                    </types>
                    <units>
                        <unit name="boolean" tID="boolean"/>
                    </units>
                </part>
            </parts>
        </scenarioType>
    </scenarioTypes>
    <relationshipTypes/>
    <parameterTypes>
         <parameterType dataType="double" defaultValue="10" desc="Tooltip/help for
         cancellation level" name="Cancellation Level"
         tID="CancelReasoningFrameworks::P_CancelLevel"/>
         <parameterType dataType="List" defaultValue="15" desc="Tooltip/help for "
         name="the list of responsibilities (%)"
         tID="CancelReasoningFrameworks::P_ListOfCollaboratingProcess"/>
    </parameterTypes>
    <responsibilityParameters>
               <parameterType tID="CancelReasoningFrameworks::P_CancelLevel"/>
         <parameterType tID="CancelReasoningFrameworks::P_
     ListOfCollaboratingProcess"/>
    </responsibilityParameters>
<model />
</rf>
```

References

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

[Bachmann 03] Bachmann, Felix; Bass, Len; & Klein, Mark. *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

[Bachmann 05] Bachmann, F.; Bass, L.; Klein, M.; & Shelton, C. "Designing Software Architectures to Achieve Quality Attribute Requirements." *IEE Proceedings: Software 152*, 4 (August 2005): 153-165.

[Bass 01] Bass, Len; John, Bonnie E.; & Kates, Jesse. *Achieving Usability Through Software Architecture* (CMU/SEI-2001-TR-005, ADA393059). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. http://www.sei.cmu.edu/publications/documents/01.reports/01tr005.html

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

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

[Buschmann 96] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; & Stal, Michael. *Pattern-Oriented Software Architecture: A System of Patterns, Volume 1*. West Sussex, England: John Wiley & Sons Ltd., 1996.

[Friedman-Hill 03] Friedman-Hill, Ernest. *Jess in Action: Java Rule-Based Systems*. Greenwich, CT: Manning Publishers, 2003.

[John 04] John, Bonnie E.; Bass, Len; Sanchez-Segura, Maria-Isabel; & Adams, Rob J. “Bringing Usability Concerns to the Design of Software Architecture,” 1-19. *Engineering Human Computer Interaction and Interactive Systems: Joint Working Conferences EHCI-DSVIS 2004*. Hamburg, Germany, July 11-13, 2004. New York, NY: Springer Verlag, 2004.

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.			
1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 2005		3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Elements of a Usability Reasoning Framework			5. FUNDING NUMBERS FA8721-05-C-0003
6. AUTHOR(S) Jinhee Lee, Len Bass			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2005-TN-030
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES			
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			12B DISTRIBUTION CODE
13. ABSTRACT (MAXIMUM 200 WORDS) This technical note brings together two different threads of work: (1) investigating the relationship between usability and software architecture that has generated a number of usability scenarios with implications for software architecture and (2) developing an architecture design assistant, Architecture Expert (ArchE). One key element of ArchE is that quality attribute knowledge can be encapsulated into *reasoning frameworks*, and a Carnegie Mellon University Master of Software Engineering project team has developed an ArchE reasoning language (ARL) with which to specify the actions of reasoning frameworks within ArchE. This note describes an ARL implementation of two usability scenarios: (1) displaying progress feedback and (2) allowing cancel. These implementations begin to provide ArchE with the ability to reason about aspects of usability that have software architecture implications.			
14. SUBJECT TERMS ArchE, ARL, usability, reasoning framework			15. NUMBER OF PAGES 68
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