The Application of the Value-Added Activity Model for the Mark-6 LE Integration Project

by

Joanna Liang

B.S. Computer Science
Massachusetts Institute of Technology, 2004

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE AT THE MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2005

Copyright ©2005 Joanna Liang. All rights reserved.

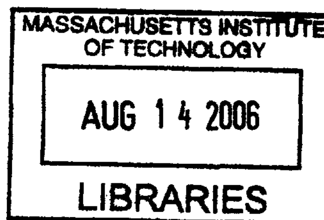
The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part.

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science

Certified by ______________________________
Paul Thordarson
Charles Stark Draper Laboratory
Thesis Supervisor

Certified by ______________________________
Professor Barbara Liskov
Ford Professor of Engineering
Thesis Advisor

Accepted by ______________________________
Professor Arthur C. Smith
Chairman, Department Committee on Graduate Theses


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
AUG 14 2006
LIBRARIES

ARCHIVES

[This page is intentionally left blank]

The Application of the Value-Added Activity Model for the Mark-6 LE Integration Project

by

Joanna Liang

Submitted to the Department of Electrical Engineering and Computer Science on May 19, 2005, in partial fulfillment of the requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Powerful information and workflow management tools can minimize risks and maximize productivity for a project. However, a conventional task-based project management approach does not provide the kind of details necessary to support key decision making processes. This thesis explores a new value-centric abstract work model, the Value-Added Activity Model, and applies it to the complex electrical and mechanical integration context of the Mark-6 LE project. We expanded the existing model to include the notion of outcomes and implemented a concrete work model to support the development of the integration application.

Thesis Supervisor: Paul Thordarson
Principal Member of the Technical Staff, Charles Stark Draper Laboratory

Thesis Advisor: Professor Barbara Liskov
Ford Professor of Engineering, M.I.T.

[This page is intentionally left blank]

ACKNOWLEDGEMENT

May 19, 2005

This thesis was prepared at The Charles Stark Draper Laboratory, Inc. under contract #N00030-05-C-0007.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

(Author's signature)

[This page is intentionally left blank]

Table of Contents

1. Introduction 9

2. Value-Added Activity Model 12
    2.1 Deliverable Agreement 13
    2.2 Commitment Protocol 14
    2.3 Dependency Management 16

3. Outcomes 18
    3.1 Design 18
        3.1.1 Outcome 18
        3.1.2 Workflow Outcome 21
        3.1.3 Workflow Execution Model 21
        3.1.4 Outcome Set 26
    3.2 Implementation 29
        3.2.1 Outcome 29
        3.2.2 Workflow Outcome 31
        3.2.3 Outcome Set 35

4. Mark-6 LE Integration Model 38
    4.1 Design 38
        4.1.1 Procedure 38
        4.1.2 Parameter 40
        4.1.3 Instantiated Procedure 41
    4.2 Implementation 43
        4.2.1 Procedure 43
        4.2.2 Parameter 45
        4.2.3 Instantiated Procedure 45
        4.2.4 Procedure Workflow Definition Language 47
        4.2.5 Procedure Loader 50

5. Discussion 51

6. Conclusion 53

[This page is left intentionally blank]

1. Introduction

The complexity of today's projects requires powerful information and workflow management tools to maximize productivity and success. The need for workflow management is driven by businesses that realized the benefits of organizing work around projects and the critical need to communicate and coordinate work across different departments. However, effective workflow management is no small task. A quick search on a Google search engine turns up a myriad of companies offering various project management solutions. While the specific design and implementation may differ, superior workflow management tools all share the common characteristic of having a good abstract work model. The thesis explores a new value-centric abstract work model, the Value-Added Activity Model, and applies it to the complex electrical and mechanical integration context of the Mark-6 LE project.

The integration application, as a part of the Mark-6 LE effort, utilizes the current web technologies and has the structure illustrated below. In Figure 1, the persistence service provides a platform for managing persistence in a database. Immediately above it, the Value-Added Activity (VAA) layer defines basic mechanisms for describing and managing generic workflow. The thesis extends the existing VAA model by adding to it the outcome abstraction to better reflect the types of workflow processes people encounter in real life. In addition, the thesis includes a design and implementation for the Mark-6 LE Integration Work Model above the VAA to support the integration application.

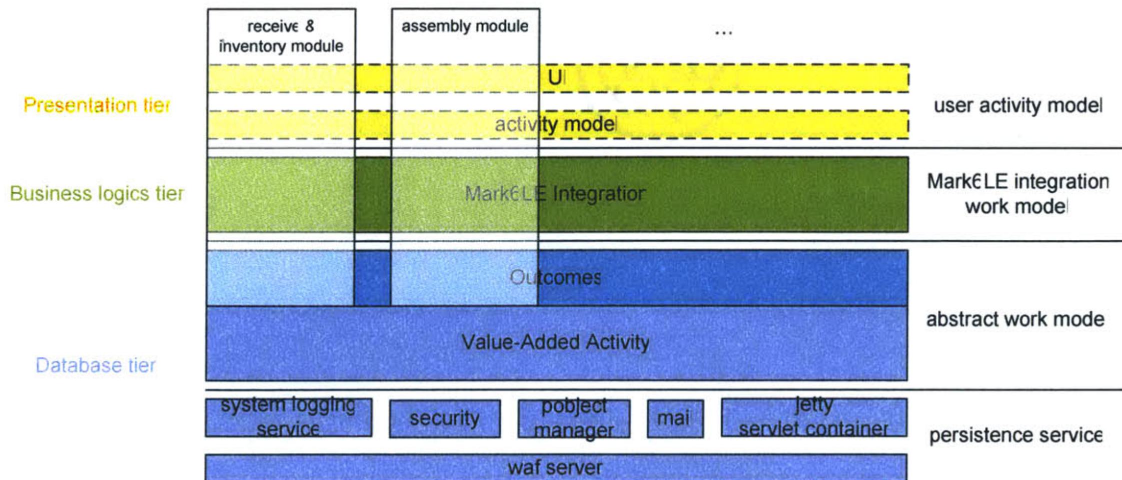


Figure 1 MarkELE integration project overview

The Mark-6 LE integration project is the first attempt to apply the VAA Model in a real-world context. In addition to building an application that will aid in a complex large-scale integration project, the thesis also provides a unique opportunity to evaluate the current VAA model. In order to be effective, the integration application has to make certain assumptions about the content and the presentation of the information in the persistent database. For example, to record the step of a worker adding a CPU to a computer on the assembly bench of the factory, the application relies on the underlying engine for finding records of all the CPU's that are currently available to the assembly bench. The study of such usage patterns can be invaluable as it demonstrates how the application is able to leverage the capabilities of the VAA and how the VAA can improve to be more useful to the application.

Lastly, the thesis takes usability into consideration when designing the integration application because it significantly affects the way the application will be perceived.

Since the integration work model sits right beneath and interacts directly with the user interface, it is important to keep user behaviors and expectations in mind when designing this layer of the system. For example, it is important to understand design-related usability issues such as system failure mode. What types of mistakes can a user make to cause a failure in the application? What should be the anticipated response from the user's point of view?

2. Value-Added Activity Model

The complexity of real-world projects often requires them to be broken into smaller and more manageable subprojects each with teams and resources that operate separately. Successful project management requires a massive effort to integrate the subprojects at various levels during the production life cycle. Communication must be streamlined across the teams and perceivable risks must be made visible at every level. Yet traditional project management methods do not provide the kind of details necessary to support key decision making processes. Conventional project management methodologies have primarily focused on managing the individual tasks within a project. The task-based approach often results in a departure from the project goal and leads to a misplaced emphasis on producing the necessary documents and reports. As the project progresses, producing these artifacts becomes an end in itself and the management is ill-equipped when faced with changes in critical decisions.

The Value-Added Activity (VAA) model was formulated in 1998 as a research and development effort to enhance project management processes. To overcome the inadequacies of conventional project management, the VAA model emphasizes project deliverables over project tasks. As opposed to the traditional task-based approach, this deliverable-based model recognizes the fact that a successful project is one that ultimately adds value to the customer. Therefore, any management or decision making processes should focus on value discussions. While it is difficult to predict the exact value impact of a decision and unforeseen circumstances can often influence that

prediction, deliverables are directly correlated to the project value, not tasks. Therefore, they are a more relevant measure of risks and successes.

2.1 Deliverable Agreement

Central to the VAA is the concept of *deliverable agreement* (DA). It is the embodiment of the participating parties' mutual understanding of the project and its deliverables. More specifically, it is an agreement between the client and the supplier of the requirements for the final product, the resources the client is expected to provide and the constraints on how and when the product should be delivered. Typically, the client has the role of laying out the requirements – what should the final product look like – and the supplier has the role of determining the necessary resources – how much money is needed to make the product. Note that the client and supplier of a project are not always the manager or executive of the respective party. It is important that key decisions be made by those most directly responsible for the task. Decision making processes in the VAA model will be discussed in more details in the next section. The figure below illustrates the basic structure of a DA.

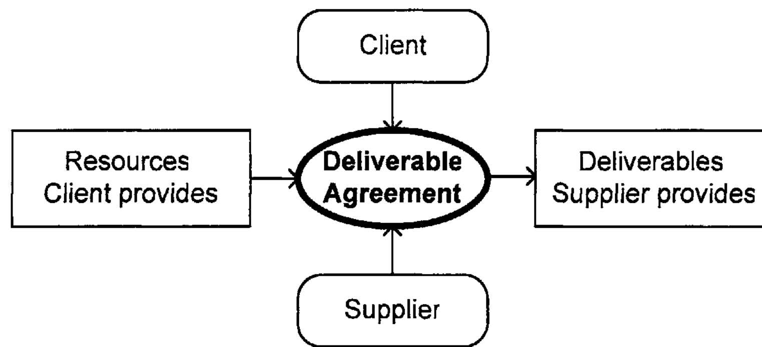


Figure 2 Basic structure of a deliverable agreement

2.2 Commitment Protocol

Projects are typically born out of market opportunities. A company discovers a popular consumer need and seeks contractors with the skills to build the product. Negotiations will then take place to determine the requirements of the project, which may include but are not limited to product specifications, expected cost, and deliverable timeline. The result of the negotiations will be the project proposal. It is important to point out that the proposal represents a *commitment* on the part of the client to provide the needed resources for the project and on the part of the supplier to produce the specified deliverables by the due date [1]. Commitment has a defined role and meaning in the VAA model. Some of the difficulties faced by conventional project management have been traced back to a lack of commitment visibility. Often time, "I will give it my best shot" is taken as a commitment and the project suffers when the person is unable to complete the task on time. Therefore, the VAA model establishes a clear commitment protocol to address this problem.

The commitment protocol is built into the life cycle of a DA and contains three key decision gates: begin, commit, and done. The DA *begins* when both the client and the supplier agree that a value opportunity exists (i.e., there is something the client wants that the supplier can provide). The DA *commits* when both parties decide to proceed with the project. The form of the commitment can be as simple as a verbal agreement between two people or as complex as a business proposal between two companies. Lastly, the DA is *done* when the client receives the deliverables and verifies that the deliverables meet the requirements. It is important to know that achieving commitment at each stage should be an iterative process. Past project management experiences have shown that initial expectations are not always realistic. Unforeseen circumstances may necessitate changes in the project requirements, budget, or timeline. They may even terminate the project all together. Therefore at each decision gate, there is an option for both parties to un-commit and go back to the previous stage. The figure below summarizes the decision processes using a typical software development project as an example. In particular, we use cost and budget as a measure of the resources needed by the project.

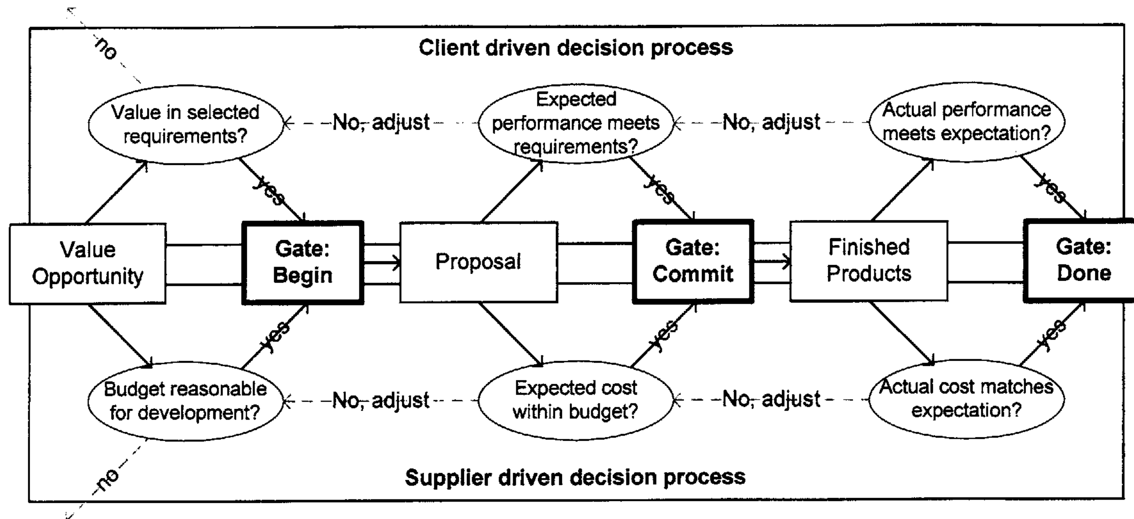


Figure 3. Commitment protocol in Value-Added Activity Model

2.3 Dependency Management

Often time, projects are linked together in a chain. In Figure 2, the deliverables produced by one DA are often the resources needed by another DA. Furthermore, any delay or change in status of one of the subprojects will inevitably impact the ability for the project to become successful. Therefore it is important that subproject dependencies be made visible to ensure that the parties can make informed decisions. As a result, a DA may be linked to other DAs that it either needs something from or it provides something for. Introducing dependencies also enables the VAA model to accommodate any size project. In addition, the VAA installs each DA with a notification mechanism so that status updates are propagated to other DAs and the respective clients. The figure below illustrates the basic dependency model as discussed in the section.

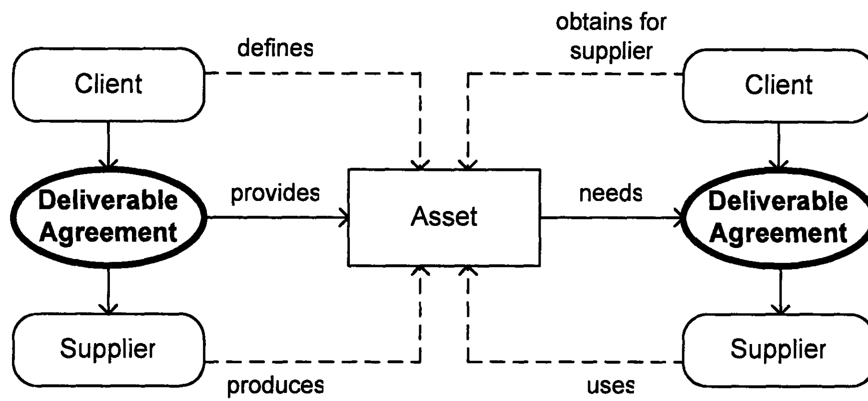


Figure 4 DA dependencies in Value-Added Activity Model

3. Outcomes

3.1 Design

The outcome abstraction is added to the VAA model to better reflect the characteristics and behaviors of real-world projects. In the next few sections, we will discuss the design of outcomes, more specifically, their relationship with DAs and assets, the mechanism for ordering them, and the logic for executing them.

3.1.1 Outcome

In order for the VAA model to be applicable to real-life projects, it is important that the model contains knowledge about the things or *assets* of the real world. After all, work is simply the manipulation of assets. If work involves the manipulation of assets, then the result or *outcome* of the work is the changes in the state of the assets. It is important to make the distinction between the outcome of a work and the assets it produces. A house painting project has the intended outcome that the color of the house is changed from its original color to the desired color. When the project is applied to a cottage, the resulting asset is a painted cottage. When the project is applied to a mansion, the resulting asset is a painted mansion. Even though the resulting assets are different, the project achieves its goal in both cases. In other words, the success of a project is not judged by the assets it produces, but whether or not it achieves its intended outcome. What the project actually

delivers is not the physical assets (i.e. a painted cottage or a painted mansion), but rather the result that the assets in question are in the desired states and orders (i.e. the color of the house is painted to the specified color). Similarly, a project that builds on the work of a previous project is dependent on the outcome of that previous project. Therefore, we extend the previous DA model to include these notions of outcomes and assets.

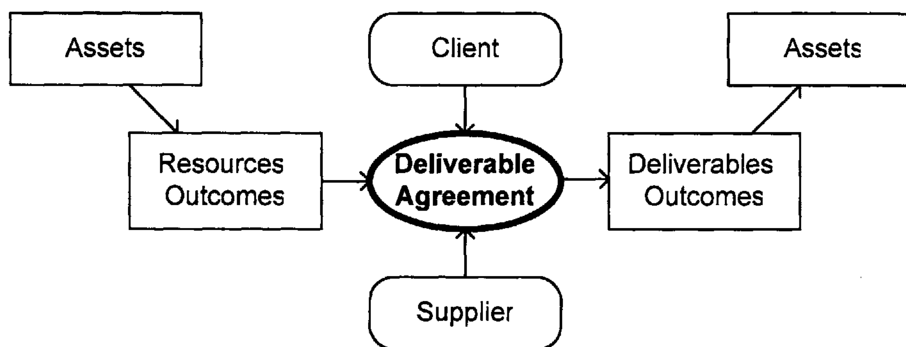


Figure 5 Extended structure of DA with outcomes and assets

As shown in the above figure, a DA takes the outcomes that another DA produced as its resources and produces another set of outcomes as its deliverables. Directly linked to the outcomes are the assets that are modified during the process. Note that an outcome may involve modifying many assets, and an asset may be affected by the work of different outcomes at different times. The relationship between outcomes and their assets is hardwired to the outcome class to allow the application to track the activities that have been performed to a particular asset. This information is especially useful when diagnosing failures. Suppose that a piece of lab equipment has failed unexpectedly. A record of all the different ways this piece of equipment has been used may shed light on the possible causes. In addition, past test and repair records could help point the

investigative team in the right direction. Without making the relation to outcome a fundamental property of the asset, compiling this information would be very difficult.

When we consider any manufacturing workflow, there are some standard processes such as assembly, disassembly, testing, etc. Although the processes vary, there are only a limited number of operations that one can perform on an asset. An operator can create an asset, insert/plug-in/connect an asset to another asset, remove/unplug/disconnect an asset from another asset, or record some changes in the status of the asset. Based on this observation, outcomes can be categorized into the following four classes: *creation*, *insertion*, *removal,* and *state change.* Note that even though assets can be physically "destroyed" in the real world – which corresponds to a complete dismantling of all its parts – they are never erased from the system. Instead, the system marks the asset as "expired" to indicate that it no longer exists in the physical world. This design is to support history tracking in the system.

Since a project's outcomes can vary greatly depending on the particular circumstances, the outcome class's design should be made flexible to accommodate the different types of outcomes. Moreover, outcomes have temporal qualities similar to assets that must be recorded in the system. The life cycle of an outcome consists of three main stages: *pending*, *active* and *completed.* An outcome is pending when it has yet to execute, most likely waiting on some preconditions to be satisfied. Usually, this correlates to when the DA is in the "begin" stage. Once the DA is in the "commit" stage, the outcome may become active and start the execution process. The outcome is completed when it has

completed execution and this will typically push the DA into the "done" stage. The meaning of each of the stages, the details of their execution processes, and their pre- and post-conditions are left intentionally vague as they are up to the subclasses to define. However, the class should implement some default status checking methods that can be overridden by its subclasses when appropriate.

3.1.2 Workflow Outcome

This thesis is interested in a particular category of outcomes, those that are relevant to workflow management. Therefore *workflow outcomes,* a subclass of the abstract outcome class, are created to handle those properties unique to outcomes that occur in workflow processes. One important characteristic of workflow processes is that they usually happen through many intermediate steps, where each successive step depends on the success or failure of the previous step. A workflow outcome, therefore, must maintain information about which other workflow outcomes it depends on in order to reason about its status. A workflow outcome cannot start executing until all of the outcomes it depends on have completed. Certainly having the capability to handle outcome ordering requires more than just that each outcome knows which outcomes it depends on. We will explore this topic in greater details in the later sections.

3.1.3 Workflow Execution Model

Before we go further into the design and implementation details, we must first discuss the interactions between the workflow engine and the user. Since the workflow model sits right beneath and interacts directly with the user interface, it is important to keep user behaviors and expectations in mind when designing this layer of the system. The figure below shows a typical interaction between a user and the system.

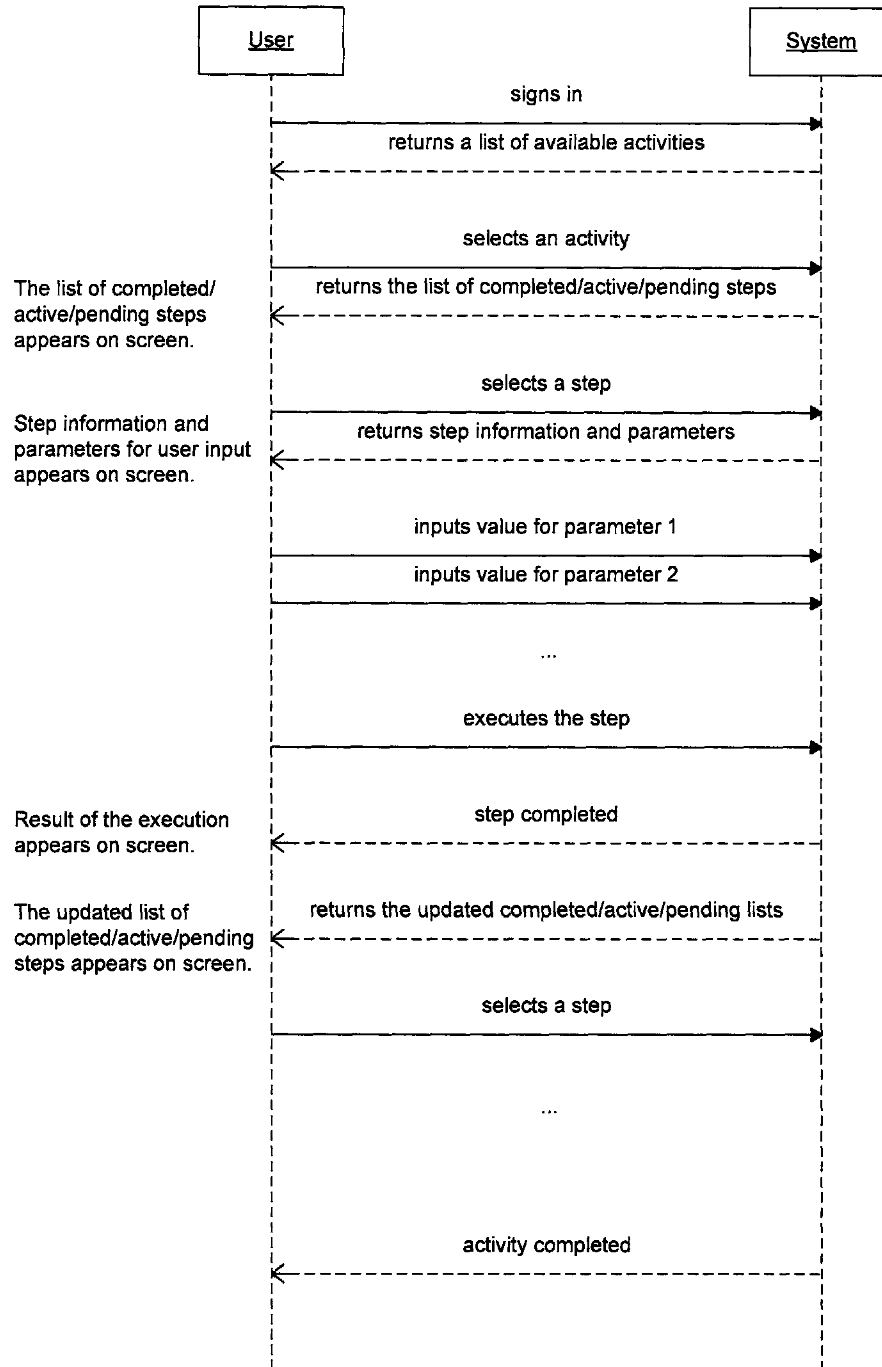


Figure 6. Workflow execution sequence

In the diagram shown above, the system displays a list of activities currently available to the user upon the user signing in. The user selects one of the available activities to start

and the system displays a list of steps pertaining to the particular activity, organized into three categories: pending, active, and completed. Only the active steps are available to the user. Then the user selects an active step to perform and the system retrieves the step information. Often time, the step will have parameterized fields that need to be filled in before it can be executed in which case the system will prompt the user for inputs. Once the user provides all the necessary parameter information, the system executes the step and displays the updated list of pending, active, and completed steps. For example, suppose that the activity is to build an iMac G5. The system will retrieve the corresponding DA from the database, and the outcomes for the DA are presented as the steps for building an iMac G5. In this example, the first available step is to install a G5 CPU to the motherboard. The user selects this step to start and now the system needs know which CPU and motherboard the user will be using. The serial numbers for the CPU and the motherboard are the parameterized fields for the outcome. They are necessary for the outcome to know which asset to modify. Once the parameterized fields are set, the outcome executes and the updated list of steps are displayed on screen.

Following the model, the four basic workflow classes are designed with a common interface for 1) verifying that a workflow outcome has been initialized properly, 2) verifying that the outcome's parameters have been set accordingly, 3) making sure that the desired state of assets is not already achieved, 4) executing the outcome, and lastly 5) reporting the result of the execution. Each of the four workflow classes contains two types of fields, fields that are set at initialization time and parameters that are filled in at execution time. Fields that are set at initialization time are checked to ensure that the

workflow outcome is formed properly. The outcome's parameters are checked right before execution to ensure that execution can proceed without failures. In addition, the outcome looks in the current database to see if the assets are already in the desired states in which case execution would be unnecessary. Lastly, the result of the execution is reported back to allow activation of pending outcomes. We will go into more details about ordering and activating outcomes in the next section.

An important note about executing outcomes: in the current design a workflow outcome collects all the required parameters before it starts to execute. This ensures that the execution can proceed without failures. For example, if the user indicates that he will be using CPU SN#cpu0001 to build the iMac G5 when in fact there is no CPU SN#cpu0001 in the factory, the system would know that something is wrong and ask the user to correct the problem before it tries to build the iMac. An alternative approach to designing the execution routine can have the workflow outcome collects parameter information and executes concurrently in many smaller intermediate transactions. This approach is significantly more complicated to implement as it requires the application to have the ability to undo partial execution. In other words, by the time the system finds out about the non-existing CPU SN#cpu0001 problem, it would have already modified the database. It would then have to undo the effects of the previous steps to preserve consistency in the database. In our design, the execution routine is done in a single large transaction. Therefore database consistency can be guaranteed without implementing any additional mechanism. On a separate note, an undo feature is desirable but not required for our application. It offers more flexibility to users but does not affect the performance

of the application. Therefore we opt for a simpler design. The implication, however, is that once executed the effect of a workflow outcome cannot be reverted unless by another outcome that specifically undoes it.

3.1.4 Outcome Set

As mentioned before, each workflow outcome maintains information about which other workflow outcomes it depends on. But that alone is not enough for the system to correctly reason about the status of a workflow outcome. Additional logic is needed to manage the ordering of outcomes. Since outcomes are already tied to the DAs in the integration work model, it may appear sensible to insert that logic directly into the DA class. Recall, however, that the DA class is designed with the purpose of managing commitment. Therefore to prevent overburdening the DA class, a new subclass of outcome called *outcome set* is created to handle the dependencies between the workflow outcomes of a DA. Each DA has exactly one outcome set which manages all of its outcomes. A single-step DA will have an outcome set with only one outcome, while a multi-step DA will have an outcome set with multiple outcomes.

An outcome set manages the ordering of workflow outcomes by maintaining a private list of all the currently completed outcomes. To determine if a workflow outcome is ready to execute, the outcome set compares the outcome's dependents list against its list of completed outcomes. If the outcome's dependent list is a subset of the outcome set's completed list, then all the outcomes that this outcome is dependent on have completed

successfully. Therefore the workflow outcome is ready to execute and the outcome set updates the pending flag of the workflow outcome to indicate that. Using this mechanism, managing execution order is simple and straightforward.

There are a couple of simple rules that must be followed when ordering outcomes. First, *a workflow outcome can be added to the dependent list of any pending workflow outcome.* It is easy to see why this must be true. A pending outcome that has yet to execute does not have any impact on the state of the database. Therefore workflow outcomes can be added or removed from its dependent list without any repercussion. Secondly, *no workflow outcome can be added to the dependent list of another workflow outcome that is either active or completed.* Certainly a pending or active outcome cannot be added to the dependent list of an already completed outcome as it would violate the dependency model (i.e. a workflow outcome is only allowed to execute after all of the outcomes it depends on have completed). Following the same line of reasoning, an active outcome cannot have a pending or active outcome added to its dependent list. Allowing such would require the outcome to abort its execution and undo the operations it had made, something that the current design does not accommodate. In the case of adding a completed outcome to the dependent list of another completed outcome, if the latter outcome was able to complete without the work of the first outcome, then it does not depend on it. Therefore it does not make much sense to have that outcome added to its dependent list.

In the previous section, we talk about how workflow outcomes must report the result of their execution in order to activate the next set of outcomes. With the outcome set, we can now discuss outcome activation. After a workflow outcome finishes executing, it informs the outcome set of the result. If execution completed successfully, the outcome set updates the status of the workflow outcome and adds it to its list of completed outcomes. Next, it goes through all of the still pending outcomes and determines if any of them is ready to execute using the algorithm discussed before. A outcome that is ready to execute will have its pending flag set to false and will appear on the active list the next time the list is compiled. When all the outcomes in an outcome set finish executing, the outcome set reports the result back to its related DA. Depending on the specific DA implementation, it will send out notifications to the appropriate parties.

At this point, we have gone through all the design details. We can revisit the sequence diagram in Figure 6 and expands it to include the interactions between DA, outcome set, and workflow outcome. Here is the complete sequence diagram.

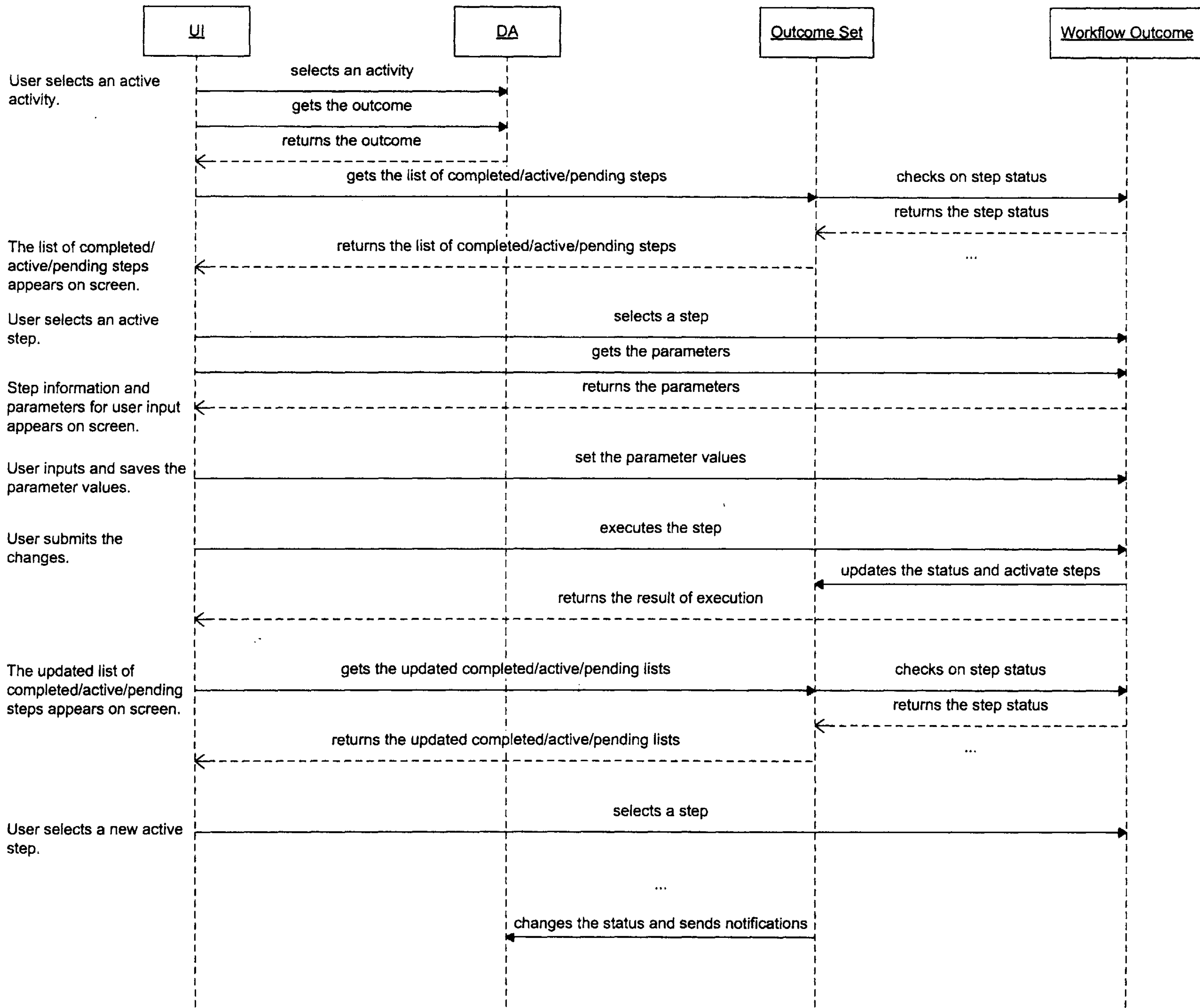


Figure 7. Complete workflow execution sequence diagram

3.2 Implementation

We have touched on all the important design issues in the previous section, the rest of the chapter will be dedicated to the implementation of outcome, workflow outcome and outcome set. In addition, we will go into more detail about each of the workflow outcome subclasses and discuss the requirements for executing them.

3.2.1 Outcome

The Outcome class is implemented as an abstract class, and as mentioned before, is expected to be heavily sub-classed to reflect the variety of outcomes possible. In addition to the standard *name* and *description* fields, the class has three more fields: an *effective date*, an *expiration date* and a Boolean *pending* flag. The first two fields, effective date and expiration date, are the results of implementing the *HasEffectiveLifeTime* interface. The effective date specifies when an outcome is created, the expiration date specifies when it completes, and together they form the lifetime of the outcome. Both the effective date and the expiration date are defaulted to null. In the current implementation, the effective date is set in the outcome's initialization routine. The expiration date is not explicitly set in the abstract class, but for workflow outcomes, it is set by the outcome set at the end of the execution routine

The Boolean pending flag is used to compute the status of the outcome. By default the flat is set to false – the outcome does not depend on any other outcome and hence is

ready to execute. Although the algorithm by which we compute the status of an outcome may vary depending on the type of outcome, the Outcome class does provide default implementation of the three status checking methods, *isPending()*, *isActive()*, and *isCompleted()*, based entirely on the outcome's expiration date and the pending flag. An outcome is pending if the value of pending is true; it is active if the value of *pending* is false and its expiration date is null; it is completed if the value of pending is false and the expiration date is a valid date. These methods should be overridden by the subclasses when appropriate.

To facilitate history tracking, the Outcome class defines a two-way any-to-many relationship between an outcome and the assets it modifies as an outcome may modify many assets and an asset may be affected by the work of different outcomes at different times. The relationship name from an outcome to its asset is *modify* and the relationship name from an asset to its outcome is *modified by*.

In addition to the utility methods, isPending(), isActive() and isCompleted(), as discussed before, the Outcome class defines two more abstract methods that are to be implemented by its subclasses: *isUsable()* and *isExecutable()*. Both methods take no argument and returns either true or false. The method isUsable() should contain logics that determine if an outcome is formed and initialized properly to be used in a workflow. Therefore, it should be called at the end of the initialization routine. Although the specifics vary depending on the subclass that implements it, at the very least it should check that all the required fields of the outcome have been initialized with legal values. The isExecutable()

method contains logics that determines if an outcome can be executed. This is different from the dependency test performed by an outcome set to make sure that a workflow outcome does not start executing before all of the outcomes it depends on have completed. The isExecutable() method involves checking that all the parameterized fields of the outcome have been set properly to ensure that execution can be carried out. Therefore it should be called at the beginning of the execution routine.

3.2.2 Workflow Outcome

The WorkflowOutcome class defines the properties unique to outcomes that occur in workflow processes. It inherits from the Outcome class and is implemented as an abstract class as the exact behaviors of workflow outcomes can vary a great deal depending on the type of workflow processes they belong to. One distinguishing characteristic of workflow processes is that they usually consist of many intermediate steps that depend on each other's result. It is therefore imperative that each workflow outcome maintains a list of the outcomes that it depends on in order to reason about its status. The dependent list is implemented in the WorkflowOutcome class as a one-way any-to-many relationship. The relationship name from a workflow outcome to the workflow outcomes it depends on is *isActiveAfter* since the outcome can only become active after all the outcomes it is tied to by this relationship. The class also defines a finder method *findOutcomeSet().* Since each outcome set is initialized with a unique name, the method takes a name string as its argument and returns the matching workflow outcome.

An equally important relationship is the one between a workflow outcome and its outcome sets. Although workflow outcomes maintain their own dependent list, in order to correctly reason about its status, it relies on its outcomes sets which have specialized logic for managing their ordering and activation. This relationship is necessary for a workflow outcome to report its result at the end of its execution routine and for the outcome set to update the status of its outcomes. The relationship is defined in the WorkflowOutcome class as a two-way any-to-many relationship. An outcome set will often have many outcomes to manage and an outcome may be a part of different DAs each with its own outcome set. The relationship name from a workflow outcome to its outcome sets is *outcomeOf*, and the relationship name from an outcome set to its workflow outcomes is *outcomes*. There is a corresponding finder method *findOutcomeOf()* for the relationship. Same as before, it takes a name string as its argument and returns the matching outcome set.

Lastly, recall that in the previous section we discuss the two rules that must be followed when ordering workflow outcomes to preserve database consistency. The two rules are 1) a workflow outcome can be added to the dependent list of any pending workflow outcome and 2) no workflow outcome can be added to the dependent list of another workflow outcome that is either active or completed. The logic that checks and makes sure that these rules are not violated is in the *addWorkflowOutcomeDependency()* method of the WorkflowOutcome class. Like the default add method *addIsActiveAfter* puts in by the persistent relationship engine, it takes one argument – the workflow outcome to be added to isActiveAfter collection. It is important that the check is performed every time

the system attempts to modify an outcome's dependent list, therefore this method should be called instead of the default addIsActiveAfter() method.

To better represent the type of workflow processes typical to a manufacturing project like the Mark-6 LE Integration Project, the WorkflowOutcome class is further extended into four subclasses: *Creation*, *Insertion*, *Removal* and *StateChange*. Each of the classes are distinguished by its set of *fields* – object fields set at initialization time and *parameters* – object fields set at execution time. In addition, the class Home interface defines specialized *execute()* and *inorder()* methods. The inorder() method checks the state of the assets in question to determine if they are already in the desired arrangement. If not, the execute() method performs the necessary modifications. In the next few sections, we will discuss in more details each of the four workflow subclasses.

Creation

The Creation class provides an abstraction for workflow outcomes specialized in the creation of assets. It has one field *assetClass*, which is set when the application defines the object, specifies the class of asset the object will create and by default is an empty string. The value of the field is verified in the overridden isUsable() method. In order to create the asset, the class has three more parameters that must be filled in: *assetName, assetRefId* and *assetDescription.* As the names implied, they specify the name, reference ID and description of the asset respectively. The values of the parameters are checked by

the overridden isExecutable() method at the beginning of the execute() method to ensure that the object has enough information to create the asset successfully.

Insertion/Removal

The Insertion class and the Removal class provide an abstraction for workflow outcomes specialized manipulating the role relationships between assets. The Insertion class has five fields: *parentAssetClass, partAssetClass, roleClass, roleName* and a Boolean field *exclusive.* Like the names suggest, the parentAssetClass field specifies the class of the parent asset, the partAssetClass field specifies the class of the part asset, the roleClass field specifies the class of the role, the roleName field specifies the name of the role and the Boolean exclusive flag indicates if the role is exclusive. If the role is exclusive, then only one asset can fulfill that role. Therefore in addition to establishing the relationship, the object must also retire any currently active ones (i.e. marking the roles as expired). The exclusive flat is by default false. The Removal class has all the same fields as the Insertion class, except for the exclusive flag. Since the object is designed to retire a currently active role-fulfillment relationship between assets, it does not need to know whether or not the role is exclusive. Both the Insertion and Removal classes have two fields: *parentAssetId* and *partAssetId.* They are the persistent database ID and are used by the Insertion/Removal object to find the assets it will operate on.

State Change

The Creation class provides an abstraction for workflow outcomes specialized in the modifying the state of an asset. It has two fields: *assetClass* and *stateClass*. The assetClass specifies the class of the asset to which a new state will be attached, and the stateClass specifies the class of the new state for the asset. In addition, the StateChange class defines four parameters to be filled in before execution: *assetId, stateName, stateDescription* and *stateValue.* As before, the assetId is the persistent databae ID assigned to the asset. The stateName, stateDescription and stateValue are used to create the appropriate state object.

3.2.3 Outcome Set

The OutcomeSet class is created specifically for managing the execution order of the workflow outcomes of a DA. It is implemented as a concrete subclass of the abstract Outcome class. The relationship between a DA and its outcome set is defined in the OutcomeSet class as a two-way any-to-any relationship. Each DA has exactly one outcome set and each outcome set belongs to exactly one DA. The relationship name from a DA to its outcome set is *outcomeSet* and conversely the relationship name from an outcome set back to its DA is *outcomeSetOf.* A DA relies on its outcome set to keep track of its internal status (i.e. how many workflow outcomes have finished executing). Therefore the class also defines three utility methods for finding out the status of its workflow outcomes. They are *findCompletedOutcomes(), findActiveOutcomes()* and *findPendingOutcomes().* The methods simply go through the list of outcomes and inquire about the status of each one.

Another important relationship defined in the OutcomeSet class is the outcome set's list of completed workflow outcomes. As discussed before, although each workflow outcome maintains its own dependent list, in order to correctly reason about its status, it relies on the outcome set which has specialized logics for managing outcomes ordering and activation. The specialized logics take the form of a list of completed outcomes which is compared against a workflow outcome's dependent list to determine if the outcome is ready to execute. The relationship is defined in the OutcomeSet class as a one-way any-to-many relationship. The relationship name from an outcome set to its list of completed outcomes is *completedOutcomes.*

As mentioned before, the routine of comparing an outcome set's completed list against its workflow outcomes' dependent lists takes place every time one of the outcomes finishes executing. It should also take place after every ordering or de-ordering event. For example, a de-ordering event – removing an outcome from another outcome's dependent list – could change the status of the latter outcome from pending to active. The routine of looking through the list of workflow outcomes and activating the appropriate ones is implemented in the *_activateOutcomes()* method of the OutcomeSet class. It is called in the Home interface methods: *orderWorkflowOutcomes(), deorderWorkflowOutcomes* and *completeExecuteWorkflowOutcome().* As the name imply, orderWorkflowOutcomes() take the outcome set, the first workflow outcome and the second workflow outcome as its arguments. It adds the first outcome to the second outcome's dependent list, then calls _activateOutcomes() to update the status of all the outcomes.

deorderWorkflowOutcomes() works in similar fashion except it removes the first outcome from the second outcome's dependent list.

CompleteExecuteWorkflowOutcome() contains the logic for handling the completion of a workflow outcome. The method is called by the workflow outcome at the end of its execute routine. If the workflow outcome completed successfully, completeExecuteWorkflowOutcome() updates the status of the workflow outcome to "completed" and adds all the assets referenced by the workflow outcome through the modify relationship to its modify list. Of course _activateOutcomes() is then called to find outcomes that are now ready to execute. If all the outcomes have completed, the outcome is reported back to the related DA and appropriate actions will be taken based on the specific DA implementations.

4. Mark-6 LE Integration Work Model

4.1 Design

The Mark-6 LE Integration Work Model is built on top of the VAA to support the type of workflow processes specific to the integration project. The next few sections will cover the design of procedures, an important concept in project planning, the use of parameters and the requirements for instantiating a workflow process.

4.1.1 Procedure

The complexity of real life projects often requires them to be broken into smaller and more manageable subprojects. Each subproject has its own deliverables that contribute to the production of the deliverables for the whole picture. A house building project involves, for example, architects who transform the home owner's vision into a blueprint, builders who constructs the house using the blueprint, electricians who wire the house based on the wiring plan, etc. This is, of course, not the complete picture. More often than not, each subproject operates according to its own set of schedule and budget. They may even have different clients and suppliers. All these factors contribute to the increasing difficulty in managing large scale projects. The VAA model attempts to solve the problem by providing a basic framework for managing project dependencies. Each DA is equipped with a basic commitment protocol and notification supports to facilitate

the process. The Mark-6 LE Integration Work Model leverages this capability of the VAA model and extends it to better fit the type of workflow processes relevant to the Mark-6 LE project.

Now pause and consider what the user's mental model of a workflow process is. Typically in a manufacturing plant, workflow processes take place on many different benches each handling a specific aspect of the work. A disassembly process, for example, involves disconnecting the parts, testing the parts, and shelving the parts based on their test results. The activity at each of the benches can be broken down even further into the steps taken by the operator. As discussed in the previous chapter, workflow outcomes correspond to the atomic operations that take place in a manufacturing workflow. In our system, they are the smallest denomination of work and represent the steps performed at a single bench. DAs also fit nicely into this model. Since they are comprised of the workflow outcomes, they represent the activities (benches) in a workflow. What is the system's representation of the overall workflow process then? It is the *procedure* abstraction and it is a particularly important concept in project planning.

Although we arrive at the concept of procedure from examining the user's mental model, there are other very practical reasons for needing this abstraction. Suppose that a factory receives an order to produce five thousand iMacs G5. For each work-in-progress on the assembly line, the system needs a separate set of DAs and outcomes to document the event. However, the activities for building iMac SN#0001 are identical to the activities for building iMac SN#0002. The only difference is the actual assets involved. Therefore,

it is better to capture the definition of a workflow independent from its actual running instances. Unlike DAs and outcomes which document actual events and are created every time the process occurs, procedures define the workflow processes and are created only once in the beginning. For the iMac example, this means that instead of manually building and mapping the DAs and outcomes from scratch each time, the user can simply select the procedure and ask the system to create a new instance of it.

4.1.2 Parameter

In addition to specifying the interconnections between various DAs and outcomes, the procedure workflow definition must also define a set of *parameters* which hold the information that can be customized for each running instance of the procedure. Suppose in the previous iMac example, the order is for twenty-five hundred iMacs with 1.8GHz G5 processors and twenty-five hundred iMacs with 2.0GHz G5 processors. How should the system handle this scenario? Certainly we do not want to define two separate procedures just for a simple change in the processor type. The better approach is to extend the procedure model to allow parameterized information. The procedure should indicate everywhere in the workflow definition that processor type is a parameterized field and may be specified later. From the point of view of user-system interaction, this means that when a user selects to start a new instance of the iMac G5 build procedure, the system should ask the user to specify the type of processor he will be using and create the appropriate instance based on the input.

4.1.3 Instantiated Procedure

We have talked a lot about procedures and their "instances". We should more formally define what these instances or *instantiated procedures* are. A procedure is the general definition of a workflow process; an instantiated procedure is the procedure as applied in a particular scenario. An iMac G5 build procedure contains instructions on how to assemble an iMac G5. Its instantiated procedures handle the logistics of collecting the parameters, soliciting inputs from the user, putting together the activities entailed by the procedure, initializing each activity with the correct set of steps, and executing them accordingly. Although procedures hold the workflow definition, instantiated procedures drive the workflow processes. Therefore DAs are linked directly to their instantiated procedures and not the procedures themselves. With the instantiated procedure abstraction, the system has completely separated workflow planning from workflow execution.

Note that so far we have not explained how a workflow process is defined in a procedure. It is certainly a topic that must be addressed and we will discuss it in more details later in this chapter. Since we have completely separated workflow planning from workflow execution, the mechanism by which we define workflow processes will not significantly impact the rest of the system. A simpler implementation may choose to present everything in a simple text file and users can edit it using any ordinary text editor. On the other hand, a more sophisticated implementation may opt for an interactive graphical interface where users can drag and drop in activities and steps. For the purpose of this

thesis, we pick the simpler approach and define workflow processes in text files using standard xml syntax.

Before we move to implementation details, we should spell out more concretely the exact sequence of the events from when a user requests to start a new procedure instance to when the workflow process executes. In fact, most of it has already been discussed in the previous paragraphs, but this is a good place to summarize what we have talked about so far in the chapter. The process can be divided into five main stages. In the first stage, the instantiated procedure reads the parameter information from the workflow definition file and creates the corresponding parameter objects. Next, the collection of parameters is presented to the user for inputs. Once all the parameter values are entered, the instantiated procedure builds the net of DAs and outcomes. In the last stage, the instantiated procedure starts the execution process by going through the list of DAs and updating the status of each one. The figure below illustrates a typical interaction between a user and the system to instantiate a workflow procedure.

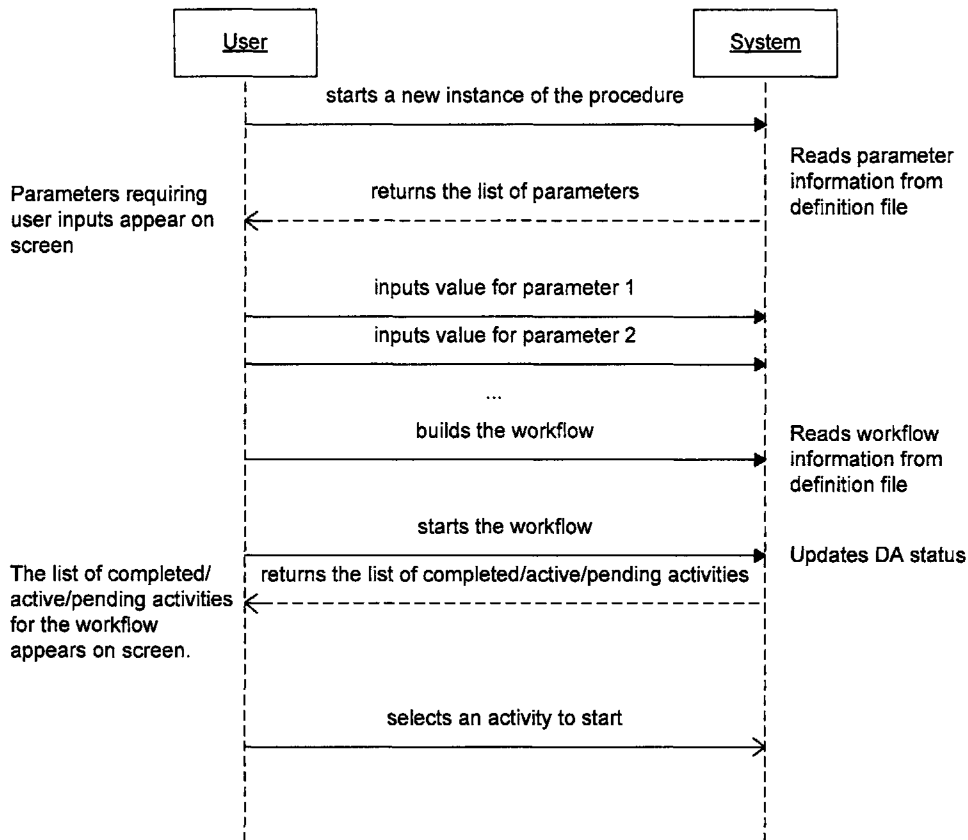


Figure 8. Procedure instantiation sequence

4.2 Implementation

Now that we have covered all the design issues related to instantiating a procedure, the rest of the chapter will focus on the implementation details of the different classes in the Mark-6 LE Integration Model. We will also describe the syntax and form of workflow definition documents in the last section.

4.2.1 Procedure

The Procedure class provides the definition for a workflow process. It is implemented as a subclass of Asset because procedures are expected to be in role-fulfillment relationships with their assets. The iMac G5 configuration item, for example, will most likely have with it a build procedure, a test procedure … etc. We have briefly mentioned before that workflow definition is written into a file using standard xml language. The exact syntax and format of a workflow definition document will be discussed later in the chapter. To attach the document file to a procedure object, the class takes advantage of the built-in capability of the underlying persistent service. To retrieve the attached document, an instantiated procedure simply calls the *getDBDocumentFile()* method on the procedure. In addition, the procedure class defines two fields to facilitate the loading of its definition file: *filepath* and *fileMimeType.* Both fields take a string as their values and are specified at initialization time.

The most important relationship defined in the procedure class is the relationship between it and its instantiated procedures. An instantiated procedure relies on its procedure to provide the workflow definition and a procedure depends on its instantiated procedures to handle the logistics involved in driving the workflow process. Therefore, the relationship is defined as a two-way any-to-many relationship. A procedure will often have more than one instantiated procedures, but each instantiated procedure can only be instantiated from a single procedure. The relationship name from a procedure to its instantiated procedures is *instantiations,* and the relationship name from the instantiated procedures back to the procedure is *instantiationOf.* In addition, the class defines the finder method

findInstantiation() which takes a name string and returns the matching instantiated procedure.

4.2.2 Parameter

The Parameter class is created primarily to be a holder for parameterized information in a workflow definition. Each parameter object is initialized with a *type*, a *name*, a *description*, a *reference name*, a *default value*, a *value* and some number of *options*. The type field specifies the type of value the parameter takes. The name field specifies a unique identifier for the parameter in the parameter database. The description field contains additional information about the parameter. The reference name field specifies the key the procedure loader uses to hash the parameter. The paper will go into more details about the use of reference name for hashing parameters in the next few sections. The default value is the value assigned to the parameter at initialization time. The value field contains the final value the parameter takes. Lastly, the parameter has a collection of options that it may display for the user to select. The collection may be null if no such predetermined list exists.

4.2.3 Instantiated Procedure

The InstantiatedProcedure class defines a single application of a procedure and contains the information and logics for driving the workflow process. While the procedure provides the workflow definition document, it is the job of the instantiated procedure to

transform the text definition into a network of DAs and outcomes in the persistent database. Therefore, the class has defined relationships with parameters and DAs. The relationship between an instantiated procedure and the parameters is a two-way any-to-many relationship. The relationship name from an instantiated procedure to its parameter is *parameters* and the relationship name from the parameters back to the instantiated procedure is *parameterOf.* The relationship between an instantiated procedure and its DAs is also a two-way any-to-many relationship. The relationship name from an instantiated procedure to its DAs is *builds* and the relationship from a DA back to its instantiated procedure is *builtBy.* The class also defines corresponding finder methods for the two relationships: *findActivity()* and *findParameter().* Similar to before, the methods take a string name and return the matching objects.

Recall the sequence diagram in Figure 8 which shows the program flow of a procedure instantiation in five stages. A point that we neglect to mention and perhaps is not clearly expressed in the diagram is that each successive stage cannot begin unless the previous stage has completed successfully. For example, the instantiated procedure cannot build the net of DAs and outcomes until all the parameters have been filled in by the user. This leads to the three stages of instantiated procedures: *buildable, built* and *executable.* An instantiated procedure is buildable when the user has inputted values for all of its parameters; it is built when it has created and wired the net of DAs and outcomes; it is executable when it has gone through the DAs and updated them to their appropriate status so that they may be started. The class defines three corresponding methods for finding out the status of an instantiated procedure: *isBuildable(), isBuilt()* and *isExecutable().*

It is perhaps not too clear why an instantiated procedure must update the status of its DAs before the process can be started. Recall that the life cycle of a DA consists of three stages: begin, commit and done. Typically, in project management language, a DA is not committed until a formal agreement has been reached by the client and the supplier to proceed with the project. This agreement usually takes the form of a proposal. However, in the case of workflow processes, there are no negotiations prior to the creation of each DA. In fact, the creation implies the agreement. Therefore if a DA does not depend on other DAs – it is ready to start – the instantiated procedure will update its status to commit. To find out the status of its DAs, the instantiated procedure class defines three corresponding utility methods: *findCompletedActivities(), findActiveActivities()* and *findPendingActivities()*. An activity is pending if its status is begin; it is active if its status is commit; lastly it is completed if its status is done.

4.2.4 Procedure Workflow Definition Language

The procedure workflow definition language specifies the syntax and format for defining a workflow process. On the top-most level, a workflow definition document has a list of parameters and a network of DAs.

```
<procedure>
      <parameter-list>
            <parameter>
            ...
            </parameter>
      </parameter-list>
      <da-net>
            <da>
```

```
            ...
            </da>
      </da-net>
</procedure>
```

Each parameter is described with a type, a name, a description, a reference name, a default value and some options. They map closely to the existing fields in the parameter class, with the exception of the reference name. This is of course not an coincident as the values in these tags are used to create and initialize the parameter object. The reference name is the unique ID used to reference the particular parameter in the workflow definition document. In the example below, *parameter 1* has the reference name *param1*.

```
      <parameter>
            <type>java.lang.String</type>
            <name>parameter 1</name>
            <description>Parameter #1</description>
            <defaultVal>param1 default value</defaultVal>
            <option-list>
                  <option>param1 option 1</option>
            </option-list>
            <refName>param1</refName>
      </parameter>
```

Inside the DA net, each DA is described with a class, a name, a description, a reference name, a list of parameters, a list of dependencies and a network of outcomes. The <class> tag specifies the type of DA to create. The DA created from the sample code below is of the type *mark6le.integration.db.da.Activity*. The <name> and <description> tags that follow initialize the newly created DA. Similar to parameters, the reference name *da2* is the unique ID used to reference this DA in the workflow definition document. Inside the <params> tag are fields of this DA that has parameterized value. We will see an example in the next paragraph. The <dependency> tag contains the reference names of all the DAs that this DA depends on.

```
<da>
      <class>mark61e.integration.db.da.Activity</class>
      <name>da 2</name>
      <description>DA #2</description>
      <refName>da2</refName>
      <params></params>
      <dependency>da1</dependency>
      <outcome-net>
            <outcome>
            ...
            </outcome>
      </outcome-net>
</da>
```

Inside the outcome net of a DA, each outcome is tagged with a class, a name, a description, a reference name, a list of parameters, a list of dependencies. Very similar to the DA block, the <class> tag in the outcome block specifies the type of outcome to create. The <name> and <description> tags contain values that are used to initialize the newly created outcome. In the sample code below, the reference name *creation2* is the unique ID used to reference this outcome in the workflow definition document. Inside the <params> block are fields of this outcome that have parameterized value. In the sample code, the only parameterized field of this outcome is *assetClass.* It takes a string as its value and is by default *asset.* The parameter that holds the value for this field is *param2.* The last tag in the outcome block is the <dependency> tag which contains the reference names of all the outcomes that this outcome depends on.

```
<outcome>
      <class>com.tgbsw.sogo.db.outcome.Creation</class>
      <name>creation 2</name>
      <description>Creation #2</description>
      <refName>creation2</refName>
      <params>
            <param>
                  <param-type>java.lang.String</param-type>
                  <param-name>assetClass</param-name>
                  <param-defaultVal>Asset</param-defaultVal>
```

```
                <param-refName>param2</param-refName>
            </param>
        </params>
        <dependency>creation1</dependency>
</outcome>
```

4.2.5 Procedure Loader

The procedure loader class has the responsibility of parsing the workflow definition document. It contains two parsers, one for parsing the parameter list and another for parsing the DA net. The reason for two separate parsers is because in the current design, an instantiated procedure makes sure that all of its required parameters are filled in correctly before it starts to build the DA net. This ensures that the DA net can be built without errors. Alternatively, the procedure loader can read the entire document once, create all the parameters, DAs and outcomes, and then update the parameterized fields later based on the user inputs. However, this approach is significantly more complicated to implement as it requires each DA to know which of its fields are parameterized. Therefore we opt for the simpler approach. This means that once the instantiated procedure starts building the DA net, no more changes can be made.

Now we have discussed all the elements of a procedure instantiation routine. We can revisit Figure 8 and expand it to include more details. Figure 9 shows the complete sequence of events in a workflow procedure instantiation.

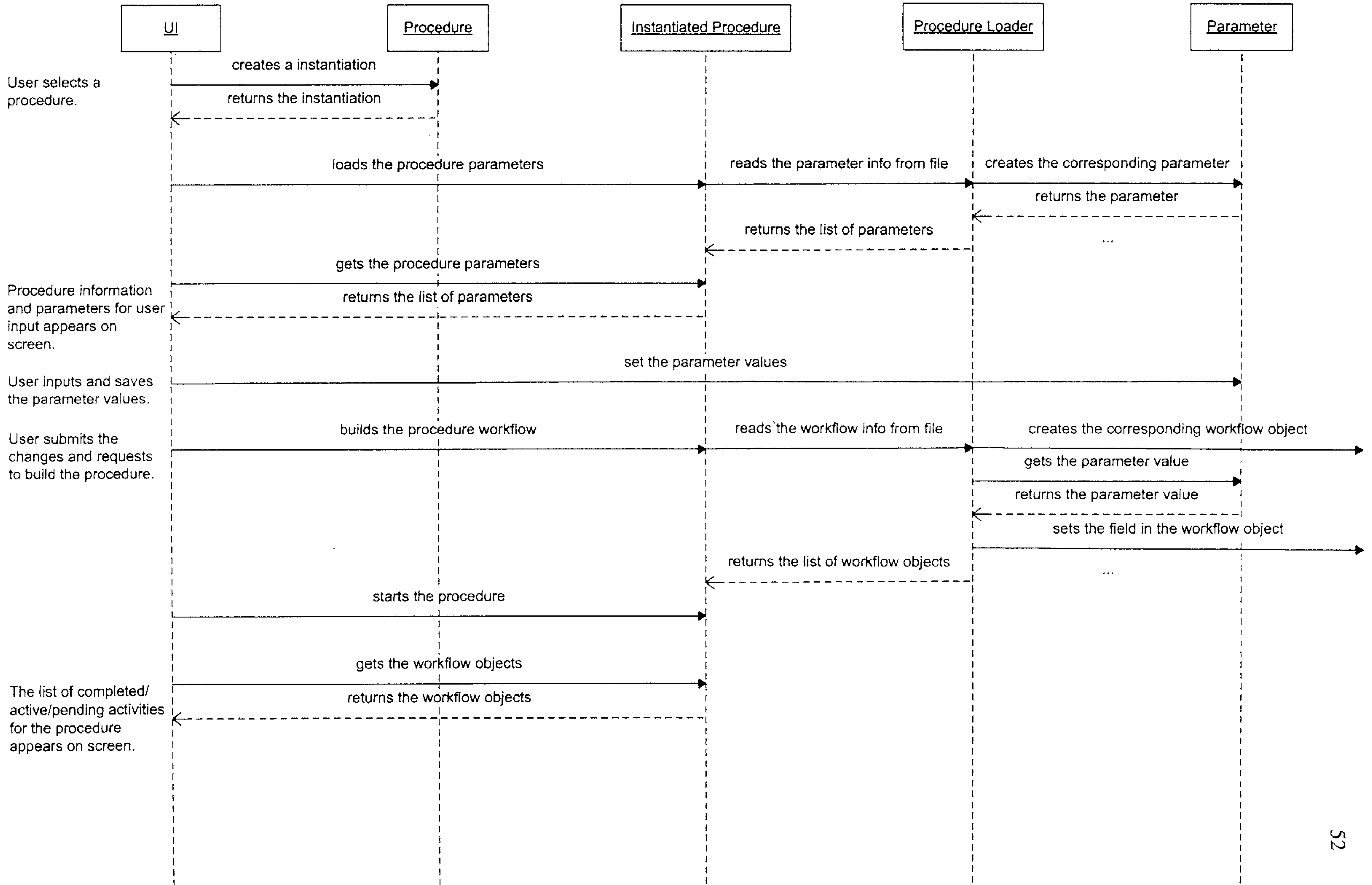


Figure 9. Complete procedure instantiation sequence diagram

5. Discussion

In the previous chapters, we have discussed the notion of outcomes in the context of he VAA model and the concept of procedures for project planning. The outcome and procedure models are based in part on the expected user's interaction with the system. Two particular cases of user-system interactions are analyzed. The first involves the process of executing a workflow from when a user selects an activity to start to when the activity completes. The second involves the process of instantiating a workflow from a procedure definition document. Together, they comprise the majority of the user tasks.

A closer examination of the workflow execution sequence diagram (Figure 6) and the procedure instantiation sequence diagram (Figure 8) finds striking similarities between the two. More specifically, a particular sequence of events is shown in both diagrams: 1) the user selects a step (procedure), 2) the system displays the parameters for the step (procedure), 3) the user inputs the values for the parameters, 4) the system executes (builds) the step (procedure) based on the parameter values. From a user's point of view, the two processes are identical except that in a workflow execution, the result is some changes in the state of assets, and in a procedure instantiation, the changes are in the activities and steps. For example, if a user requests to start the procedure to build a computer, instantiating this procedure will result in some activities and steps being created and wired together to perform the defined task. One of the steps in building a computer is to install a CPU onto a motherboard. In step 2, the user notes that he will be using CPU SN#cpu0001 and motherboard SN#mb0001. Executing this step then will

result in CPU SN#cpu0001 being attached to motherboard SN#mb0001 – a change in the state of both the CPU and the motherboard.

This indicates that perhaps outcomes and procedures should have parallel structures in our system. If we examine a workflow definition document, on the top-most level, it consisted of a list of parameters and a network of DAs. These parameters and DAs completely determines the behavior of a corresponding instantiated procedure, and while procedures in the real world can vary greatly in type and scale, they can all be captured by this simple model. This is however not true for the current outcome model. The outcome model tries to achieve the same flexibility through heavy sub-classing. Even though it is a valid approach to the problem, the different subclasses can become difficult to manage as the scope of the project expands. To mimic the procedure model, the workflow outcome class will no longer be sub-classed and the execution logic specific to each subclass will be captured by parameters instead. This means that in addition to modifying the outcome model, we must also adjust the syntax and form of the workflow definition document.

This revelation came near the end of May after we had completed the design and implementation work of the original model. However, any software development project is an iterative process and so we have included it here in the thesis as an item for future discussion.

6. Conclusion

The Value-Added Activity Model and the Deliverable Agreement paradigm provide a powerful mechanism for managing complex and agile projects based on commitment and dependency. In considering the characteristics common to real-world projects, the thesis expanded the model to include the notion of outcomes. In addition, the thesis offers a design and implementation of the Mark-6 LE Integration Model, a concrete work model that tailors to the particular tasks in the Mark-6 LE integration effort. A closer examination of the outcome model shows that it may be generalized to parallel the design of procedures in the integration model. This is an area for potential improvement and should be considered for future versions of the system.

References

[1] Haekel, S. H., Adaptive Enterprise: Creating and Leading Sense-And-Response Organizations, Harvard Business School Press, Boston, MA, 1999.