A Framework for the Specification of Acquisition Models

December 2001

B. Craig Meyers
Patricia Oberndorf

TECHNICAL REPORT
CMU/SEI-2001-TR-004
ESC-TR-2001-004

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

A Framework for the Specification of Acquisition Models

CMU/SEI-2001-TR-004
ESC-TR-2001-004

B. Craig Meyers
Patricia Oberndorf

December 2001

Dynamic Systems Program

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

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

FOR THE COMMANDER

Norton L. Compton

Norton L. Compton, Lt Col, USAF
SEI Joint Program Office

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

Table of Contents

List of Figures

List of Tables

Abstract

This report describes a framework for the specification of acquisition models. The exposition is formal in nature. The framework is defined in terms of activities, events, requirements, and instances of a system. In addition, various relations among these items, such as the relation between acquisition activities and acquisition events, are defined. The timing properties associated with the items receives special treatment.

The value of a framework is that one can develop specifications of various acquisition models, such as waterfall, spiral, or incremental, as instances of that framework. Formalizing the specification of an acquisition model has benefit in that one can reason about the characteristics of the domain addressed by the model. When this is done for multiple acquisition models, each derived from the same framework, it is possible to compare different acquisition approaches.

Executive Summary

System acquisition—the set of activities performed to procure, develop, and maintain a system—is a daunting challenge. This is particularly true for large, complex systems. There is ample evidence of routine cost and schedule overruns, not to mention acquisition failures.

As acquisition has increased in scale and complexity, new approaches have been developed to meet the challenges. Such approaches to acquisition are based on some model. For many years a waterfall model was applied (largely in a development context), although now one hears of spiral, incremental, and evolutionary acquisition models. Although such models are popular, they are often lacking in a precise specification.

Our goal in this report is to develop a specification of a general acquisition framework. The key elements of the framework are activities, events, requirements, and instances of a system. In addition, various relations among the key elements, such as the relation between acquisition activities and acquisition events, as well as timing properties, receive special treatment. We believe that from a specification of a framework, one can develop specifications of a particular acquisition model as an instance of that framework. This in turn allows one to perform a formally based comparison of various acquisition models, although to do so, we recognize it may be necessary to extend the specification of the framework to treat characteristics unique to a particular model.

The approach to the specification of the framework is mathematical in nature. The value of such a formal approach is that it provides a clear, consistent, concise specification of the framework. We believe there are additional practical benefits that can be obtained as a result of a formal specification of a framework and the specification of models based on that framework. For example, we believe it should be possible to use the framework developed here as a means to develop practical policies, procedures, and guidelines that can help acquisition managers. With the framework as a basis, it is possible to identify relevant acquisition activities and their associated events that are of interest to an acquisition manager. In addition, the timing properties concerning when acquisition activities are performed can also be addressed. But first things first.

1 Introduction

Recent years have witnessed the development of different approaches to system acquisition. For many years, a waterfall approach was common, but that was superseded by other approaches. Currently a number of different approaches are popular, including spiral, evolutionary, and incremental. However, many such approaches are only loosely specified, rather than presented on a well-defined foundation.

We define *acquisition* to be the set of activities performed to procure, develop, and maintain a system. This takes a broad view of acquisition. Some might prefer to equate acquisition with contracting activities, but we feel such a view is too limiting. Further, our concern is for the life of a system, not just from initiation to contract award, or just development.

All acquisition is performed in the context of some overall model, whether or not that model is clearly specified. The use of an explicit acquisition model helps guide an acquisition project in the manner in which the acquisition is conducted. Despite the prevalence of popular acquisition models such as those mentioned above, we have not seen a clear, precise specification of these various models. The lack of a well-specified model has numerous consequences. Among them, we mention the possible confusion that may arise when one attempts to conduct an acquisition based on a loosely specified approach.

Intended Audience

The intended audience for the material presented in this report consists of persons interested in

- how one specifies a framework that can be used to describe different acquisition models
- using the framework specified here to develop a formal specification of a particular model
- comparing different acquisition models

In addition, because the specification developed here is formal in nature, persons with an interest in formal approaches may also gain value from it. We recognize that persons who are managing an acquisition project may not approach their problems from a mathematical perspective. To aid the reader, in Appendix B we provide some background on the language and approach used in this report.

Certainly there has been much work done that is oriented toward various facets of software development processes. Examples of this include the *Rational Unified*

Process (RUP),[1] as well as development of standards for the software life-cycle process [4]. Our work differs from these, and others, in that our focus is toward formal specification of an *acquisition* framework that can then be used to develop particular acquisition models.

Acquisition Frameworks and Models

Our goal here is to develop an acquisition framework that may be applied to the specification of different acquisition models. The *acquisition framework* is a general specification of

- acquisition activities
- acquisition events
- requirements[2]
- instances of a system
- relations among the above entities, such as between activities, and among activities and events
- operational semantics (especially timing properties) arising from activities and events and their relations

The acquisition events may be *internal* or *external*, depending upon whether they are initiated within or outside the scope and control of the acquisition project.

Using the acquisition framework, it is possible to define an *acquisition model* as an instance of that general framework. We recognize that a particular acquisition model may require additional concepts and details that are not included in the framework. Inclusion of such model-specific information supplements the specification of the framework.

Use of Formal Approach

Our approach to specifying the framework will be formal in nature.[3] The use of a formal approach allows us to develop a clear, concise specification of the semantics

1. See http://www.rational.com/products/rup/index.jsp.

2. At first, the reader may find it surprising that requirements are included in the framework. The reason for inclusion of requirements will be made clear in Section 2.3.3 on page 20.

3. We will use a second-order predicate calculus that is under development and known as *Nestor*. Readers who have general familiarity with predicate calculus or with formal languages such as Z should not have difficulty reading the specification developed here. Where necessary, we will provide notes to explain the language semantics. Appendix B contains an overview of the language used to develop the specification of the framework.

of the acquisition framework. From this common framework, different acquisition models can be developed. The resulting models, also formal in their specification, gain the advantages of a formal specification of the framework, e.g., clarity and precision of specification. Further, the use of a formal specification allows us to compare different models and gain deeper understanding of the fundamental characteristics of a given model. Thus, the use of a formal approach allows us to develop a solid foundation on which to reason about the domain of acquisition.

We further believe that as acquisition becomes more complicated, so does the underlying model, though the model is often not fully defined. It is hoped that a formal approach will allow for a greater understanding of the semantics of different acquisition models. The use of a formal approach is valuable to analyze the description of a specification, whether it be for a model of a software system or a model of an acquisition.

We admit that the amount of effort to develop formal specifications is nontrivial. But this is like any other formal specification: it takes time to develop an understanding of the textual description and then to architect and cast that in a formal manner. One response to the approach taken here might be to reject it outright because it forces one to address details that one would perhaps rather not. Our reply is that acquisition models deserve the same treatment that is expected of systems: their specifications should be clear, concise, consistent, and correct. We should expect no less of the specification of an acquisition model!

Practical Considerations

Although the approach taken in this report is mathematical in its presentation, we believe there are several practical benefits. From a well-specified framework, one can develop a well-specified model for a particular acquisition approach. From a well-specified model, it should be possible to develop policies, procedures, and guidelines that are based on a sound foundation.

For example, the framework defined here includes a specification of the notion of an activity. Some practical issues that arise when managing a real acquisition project that need to be considered are:

- What are the activities that will be performed?
- How does a given activity relate to other activities?
- Are there criteria that must be satisfied in order to initiate an activity? If so, what are they?
- Are there criteria that must be satisfied in order to terminate an activity? If so, what are they?
- Can two different activities be performed at the same time?

Resolution of questions such as the above are of importance to the practical management of an acquisition. Recognition of such questions is derived from the specification of a framework and its associated instance for a particular acquisition model. In Appendix C we provide some guidance on how the framework may be used to specify a particular model. Associated with the specification of the model there can be corresponding guidelines, policies, and procedures relating to how the model is implemented.

A question related to the specification of an acquisition model is the execution of that specification.[4] That is, given the specification, how is it performed, by members of an acquisition team that may include program office staff, contractors, and other organizations? A given specification of an acquisition model can be performed in different ways. Our focus here, however, is with the development of a specification more than the execution of that specification.

Organization

This report is organized in the following manner. The following section contains a specification of the acquisition framework. In Section 3 we illustrate an application of the framework to a waterfall model. Then, in Section 4 we provide some general discussion about various acquisition approaches and the utility of a formal specification. Several topics that are possible extensions to the framework are described in Section 5. A brief summary of the report appears in Section 6.

A number of appendices accompany this report. Appendix A provides a summary of the mathematical exposition of the framework. Appendix B provides a brief tour of the language *Nestor*, which is used to develop the specification of the acquisition framework. Appendix C provides guidance for those who may be interested in using the framework developed here to specify a particular acquisition model. Appendix D provides some additional discussion concerning a specification of the waterfall model. Finally, Appendix E contains an index of the mathematical operations and data types that are present in the report.

We gratefully acknowledge discussions with colleagues Jim Smith, Patrick Place, Eileen Forrester, Rick Barbour, and Jack Ferguson.

4. A relevant analogy is the following: There is a difference between the specification of a computer program (the code) and the manner in which that specification may be executed. For example, the code can be executed on a single processor or multiple processors in parallel.

2 Framework Specification

In this section we will describe the specification of the acquisition framework. This includes specification of data types, relations, predicates, and timing properties. We will also describe some completeness requirements placed on the framework. As part of developing the specification of the framework, we will show some small examples to indicate how the framework can be used to develop particular acquisition models.

2.1 Data Specification

The development of the specification for the acquisition framework will begin by considering various data types that will be used. (Some background on data types in *Nestor* can be found in Section B.2 on page 91.)

2.1.1 Activities

To begin, we need to specify a general acquisition activity. We write this as a free type declaration:

[ACTIVITY]

We want to deal with *sets* of acquisition activities. These are declared as:

$\text{Acq\_Activities}_{[t]}$: {ACTIVITY}

The presence of the subscripted [t] denotes that the data object is dynamic. That is, the elements of the data object change in time. If a data object is declared without such a notation, it means it is static and its elements cannot change with time.

We also require that the set of acquisition activities must not be empty.[5] In other words:

$$\#\text{Acq\_Activities}_{[t]} > 0 \qquad (2.1)$$

As a brief example of how activities may be used in the specification of some acquisition *model*, the following illustrates how some activities can be declared:

5. One might argue that the set of acquisition activities should be allowed to be empty. However, were that to be the case, there would be no model!

Requirements_Management,
System_Design,
System_Test,
Risk_Management,
Contract_Management,
System_Maintenance,
System_Deployment,
Budget_Planning: ACTIVITY (2.2)

We can represent the above activities as a set by writing

$$\text{Acq\_Activities}_{[t]} = \{\text{Requirements\_Management, System\_Design, System\_Test, Risk\_Management, Contract\_Management, System\_Maintenance, System\_Deployment, Budget\_Planning}\}$$

Furthermore, suppose we were interested in separating the above activities into development activities and management activities. We can do this by declaring two sets and then requiring that they be disjoint, as follows:

$$\text{Development\_Activities}_{[t]} = \{\text{Requirements\_Management, System\_Design, System\_Test, System\_Maintenance, System\_Deployment}\}$$
$$\text{Management\_Activities}_{[t]} = \{\text{Risk\_Management, Contract\_Management, Budget\_Planning}\}$$

$$\text{Development\_Activities}_{[t]} \cap \text{Management\_Activities}_{[t]} = \varnothing$$

2.1.2 Events

The second major element of the framework is the notion of an *event*. It is important to distinguish between internal and external events. The differentiation is based on whether an event is initiated within or outside the scope of a particular acquisition project, respectively. We declare the following two basic data types:

[INTERNAL_EVENT, EXTERNAL_EVENT]

We can also declare sets of these events as:

$$\text{Internal\_Events}_{[t]}: \{\text{INTERNAL\_EVENT}\}$$
$$\text{External\_Events}_{[t]}: \{\text{EXTERNAL\_EVENT}\}$$

It is assumed that the set of external events represents those external events that are relevant to a particular application of the framework to a given model. We require that the set of internal and external events do not have any events in common. Mathematically, this requires

$$\text{Internal\_Events}_{[t]} \cap \text{External\_Events}_{[t]} = \varnothing \qquad (2.3)$$

Note that we do not place a restriction on the cardinality of the sets of internal and external events. For example, one model may deal with external events, while another may not. Hence, specifying in the *framework* a cardinality restriction on events would be considered over-specification.

To illustrate an application of internal and external events, here is an example of some internal events:

```
Requirements_Review_Scheduled,
Budget_Reviewed,
Design_Review_Initiated,
Contract_Award: INTERNAL_EVENT
```

Note that each of the above is an event that is within the scope of an acquisition project, in the sense that the project controls when the event happens, as well as what other activities may be performed in response to that event.

External events, on the other hand, are those that are initiated outside the scope of an acquisition project. The following is an example of how we could declare some external events:

```
New_Standard,
Standard_Revision,
Standard_Withdrawal: EXTERNAL_EVENT

New_COTS_Product,
COTS_Product_Upgrade_Available,
COTS_Product_Unsupported: EXTERNAL_EVENT
```

We have separated the above external events into those dealing with standards and commercial off-the-shelf (COTS) products, respectively. We do this so that we can reason about each of them independently.

2.1.3 Requirements

One characteristic of an acquisition is that the resulting system must satisfy a specified set of requirements. To develop this notion in the framework, we introduce a free type to denote a requirement, namely:

```
[REQUIREMENT]
```

A set of requirements is simply represented as[6]

$\text{Requirements}_{[t]}$: {REQUIREMENT}

We require that the set of requirements be non-empty, that is

$\#\text{Requirements}_{[t]} > 0$

Examples of how requirements may be declared are similar to the preceding discussion of events. For example, we can declare some set of requirements as follows:

Database_Requirements,
Security_Requirements,
Fault_Tolerance_Requirements: REQUIREMENT (2.4)

When we introduced the data type to represent a requirement, namely `REQUIREMENT`, we did so without further comment. Our assumption in developing the framework is that a requirement is *atomic* in nature. This is tantamount to a well-specified condition. For example, we do not assume that a requirement is contained in some other requirement. Removing the atomic-like nature of a requirement would require a different approach to the development of the framework, e.g., additional operations would be necessary to separate a given requirement into its constituent sub-requirements. In practice, requirements may not always be specified in such an ideal way; in the framework we may remove such a condition. We would argue, however, that requirements specified in an atomic-like manner should be a goal of an acquisition project.

2.1.4 System

There are many ways in which an acquired system can be realized. In particular, there can be multiple instances of a system, each with perhaps different functionality. Another choice is for there to be an instance of a system, which then evolves in time, as a result of changes to its requirements. To accommodate these various options, we simply introduce a free type

[SYSTEM_INSTANCE]

Recognizing that a given acquisition may produce, either initially or as a result of its evolution, multiple instances of a system, we represent these different instances as a set:

6. Note that the choice of a set (as opposed to a *bag*) guarantees uniqueness of the requirements; i.e., there are no duplicate requirements.

$\text{System}_{[t]}$: {SYSTEM_INSTANCE}

We do not require that the set *System* must be non-empty; that is we do not require #System > 0. The reason is that initially, the set *System* may indeed be empty.

As a simple example, consider the case where a system is being developed in a series of (incremental) builds. Suppose it is intended that there will be three builds. Each build represents an instance of the system, and we can declare

Build_1, Build_2, Build_3: SYSTEM_INSTANCE (2.5)

We emphasize that the above specification is that for data types and not functionality associated with those data types. For example, we recognize that the functionality of an instance of a system could overlap with other instances of that system. Nothing in the preceding would prevent such a functional specification from being developed.

2.2 Relations Among Basic Elements

There are a number of relations that can be defined with respect to the acquisition activities, events, requirements, and system instances that are the basic data types used in the framework. Relations help to associate, or couple, the basic data types (some background on relations in *Nestor* can be found in Section B.3 on page 94). Some guidance on the specification of these relations is provided in Table 2-1.

	Activities	**Events**	**Require-ments**	**System**
Activities	Section 2.2.1	Section 2.2.2	Not specified in framework (see Section 2.2.4)	Not specified in framework (see Section 2.2.4)
Events		Not permitted	Not specified in framework (see Section 2.2.4)	Not specified in framework (see Section 2.2.4)
Require-ments			Not specified in framework (see Section 2.2.4)	Section 2.2.3
System				Not specified in framework (see Section 2.2.4)

Table 2-1: Guide to Framework Relations

Of the possible relations one could construct, we need only be concerned with those along the diagonal or the upper diagonal indicated in Table 2-1. Note that the framework does not permit relations among events; it is assumed that events are always coupled to an activity.

2.2.1 Relations Among Activities

First, we will define a relation between two acquisition activities. The requirements that we impose on this relation are as follows:

- A given activity may be related to one or more activities.
- Every activity must be related to at least one other activity.
- An activity cannot be related to itself.

The relation satisfying the above requirements is specified below:

Activity_Relation: $\text{Acq\_Activities}_{[t]}$ <-----> $\text{Acq\_Activities}_{[t]}$
$\forall\, a_i, a_j \in \text{Acq\_Activities}_{[t]}$
• **Activity_Relation** $(a_{i\,,}\, a_j)$
$a_i \neq a_j$ (2.6)

An example of this relation is shown in Figure 2-1.

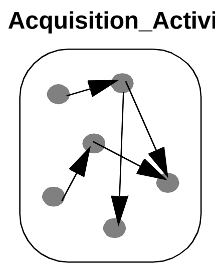


Figure 2-1: Interaction of Acquisition Activities

The characteristics of the relation, represented by the symbol “<---->”, indicate that the relation is total over both the domain and range (i.e., the relation must be defined for all elements of the set of acquisition activities). It furthermore indicates that the relation is many-to-many. In other words, a given acquisition activity may relate to multiple other activities, and, furthermore, a given acquisition activity may be related to multiple other activities.

For example, consider the activities declared in Eq. (2.2). To indicate that there is a relation between the activities requirements management and system test, we simply write:

Activity_Relation (Requirements_Management, System_Test)

As another example, suppose we wanted to declare relations between risk management and both system design and system test. We can write these as

Activity_Relation (Risk_Management, System_Design)
Activity_Relation (Risk_Management, System_Test)

In reality we would need to specify more relations. Although it is practical to do so, it is *required* by the framework in the manner in which the relation in Eq. (2.6) is declared. In particular, Eq. (2.6) requires that every activity be related to at least one other activity. This can be viewed as imposing a *conformance* requirement on all acquisition models that claim to conform to the framework.

2.2.2 Relations Among Activities and Events

The next relation that we need to specify is that involving events and acquisition activities. First, we specify this for the internal events by defining:

$$\textbf{Internal\_Event\_Activity\_Relation:}\ \text{Internal\_Events}_{[t]} \text{ <---|-> } \text{Acq\_Activities}_{[t]}$$
$$\forall\, e_i \in \text{Internal\_Events}_{[t]} \bullet \exists\, a_j \in \text{Acq\_Activities}_{[t]}$$
$$\bullet\ \textbf{Internal\_Event\_Activity\_Relation}\ (e_i, a_j) \qquad (2.7)$$

This mapping is shown in Figure 2-2.

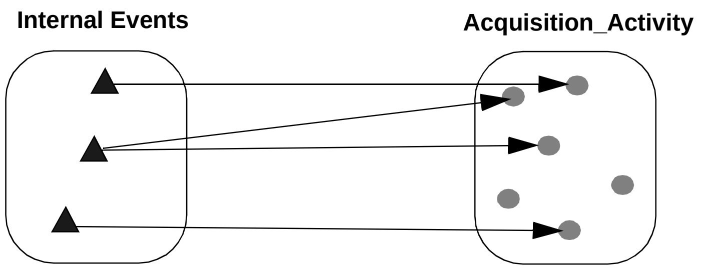


Figure 2-2: Mapping Internal Events to Acquisition Activities

The basic requirements for the mapping of internal events to acquisition activities include the following:

- Every internal event must be mapped to some acquisition activity; this is a completeness requirement.
- The same internal event can be mapped to different acquisition activities.
- Not all acquisition activities are required to be associated with some internal event (i.e., the relation is partial over the range).

For example, earlier we specified an internal event for budget review and an activity for budget planning. Using the above relation, we can declare a relation between these two as follows:

Internal_Event_Activity_Relation (Budget_Review, Budget_Planning)

Or suppose one wanted to have a budget review associated with the activity of a design review; we can declare this by

Internal_Event_Activity_Relation (Budget_Review, Design_Review)

The second group of relations included in the framework are those that relate external events and acquisition activities. This mapping is defined as

External_Event_Activity_Relation: $\text{External\_Events}_{[t]}$ <-|---|-> $\text{Acq\_Activities}_{[t]}$
$\exists\ e_i \in \text{External\_Events}_{[t]},\ a_j \in \text{Acq\_Activities}_{[t]}$
$\bullet$ **External_Event_Activity_Relation** (e_i, a_j) (2.8)

We show the character of this mapping below.

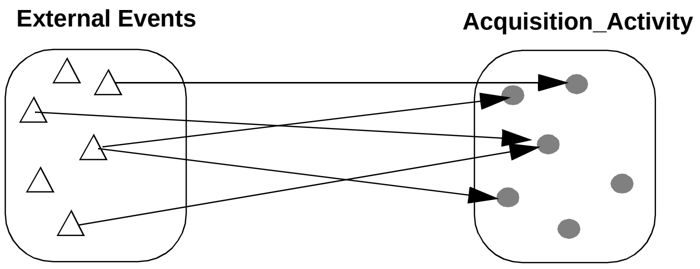


Figure 2-3: Mapping External Events to Acquisition Activities

The relation given in Eq. (2.8) is very similar to that in Eq. (2.7). However, there is an important difference. In Eq. (2.8), the relation for external events need not be defined for all possible external events that can exist outside the scope of the project. Mathematically, the relation is partial over the domain, rather than total. The choice for having the relation be partial over the set of external events simply recognizes that there may be external events that may not be of relevance to a particular acquisition project. For example, the existence of a new COTS product (an event) that is not relevant to a particular acquisition is not an external event of interest to the project.

To illustrate how the relation between external events and activities may be used in a particular model, suppose we wanted to declare a relation between a revision of a standard and the activity associated with requirements management. We can write this as

External_Event_Activity_Relation (Standard_Revision, Requirements_Management)

Similarly, we may also want to relate the existence of an upgrade of a COTS product to requirements management. This would be declared by the relation

External_Event_Activity_Relation (COTS_Product_Upgrade, Requirements_Management)

2.2.3 Relations Between Requirements and System Instances

A feature that distinguishes one instance of a system from another is the requirements a given instance must satisfy. This suggests, and practicality dictates, that there is a function that relates requirements to one, or possibly more, instances of a system. In the present case the relation must satisfy the following: it must permit that

- a given requirement may be mapped onto one, or possibly more, instances of a system
- there may be more than one mapping of a requirement onto either one or more instances of a system
- each requirement must be mapped to some instance of a system

The result of the preceding is that the mapping of requirements is many (requirements) to many (system instances). It is, furthermore, total over both the domain and range.

The mathematical representation of the preceding relation between requirements and a system is defined as

$$\begin{aligned}&\textbf{Requirements\_Mapping:}\ \text{Requirements}_{[t]} \text{ <----> } \text{System}_{[t]}\\ &\quad \forall\, r_i \in \text{Requirements}_{[t]} \bullet \forall\, s_j \in \text{System}_{[t]}\\ &\qquad \bullet\ \textbf{Requirements\_Mapping}\ (r_i, s_j)\end{aligned} \tag{2.9}$$

One example of the relation between requirements and a system is shown in Figure 2-4. Here, we show a set of requirements that are associated with one instance of a system. Note also that, in this particular case, the mapping is many-to-one; that is, each requirement is mapped to only one instance of the system.

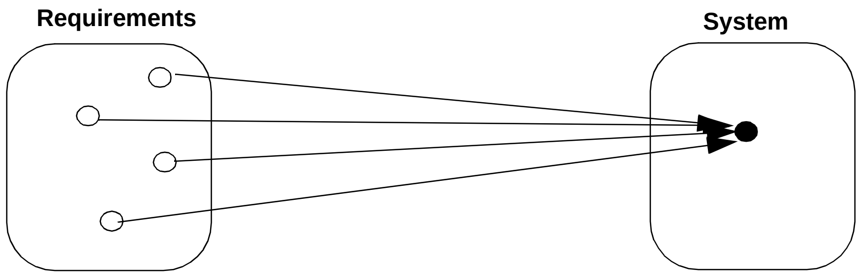


Figure 2-4: Mapping Requirements to a Single System Instance

A second example is shown below in Figure 2-5. Here, there are two instances of a system, denoted **1** and **2**. Note that some of the requirements are mapped to each instance of the system, while others are only mapped to one instance of the system.

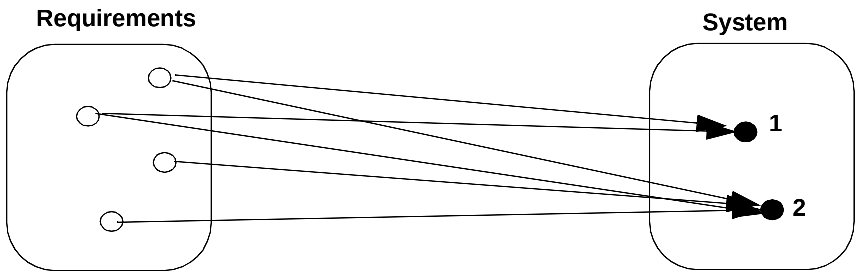


Figure 2-5: Mapping Requirements to Multiple System Instances

The relation indicated in Figure 2-5 can represent two different alternatives. First, it may be the case that there are two distinct (unrelated) systems that happen to satisfy the requirements as indicated. A second alternative is that System **1** satisfies a subset of the total requirements, which then evolves to satisfy the other requirements, resulting in System **2**. There is not sufficient information specified in the figure to determine which of these two alternatives is intended.

Let us illustrate the notion of requirements mapping. Earlier, in Eq. (2.5) we declared three builds as being instances of a system. We had also identified some requirements (see Eq. (2.5)). Suppose we wanted to specify that the database requirements are related to build 1 and the security requirements are related to build 2. We can specify these relations as:

Requirements_Mapping (Database_Requirements, Build_1)
Requirements_Mapping (Security_Requirements, Build_2)

We recognize, from a practical perspective, that the preceding relations are not complete. The reason is that we have requirements for fault tolerance, and no relation for how these requirements are related to an instance of a system has yet been specified. This illustrates some of the practical considerations of applying a general framework to a particular model with regard to completeness of requirements specification. Formal consideration of completeness criteria will be presented in Section 2.5 on page 25.

2.2.4 Additional Considerations

In developing the framework, we did not specify all possible relations involving the basic framework elements that are noted in Table 2-1 on page 9. The following provides some brief rationale for the relations that were not specified.

2.2.4.1 Activities and Requirements

A relation between activities and requirements was not included in the framework because we felt it was at a lower level of specification, and thus is more relevant perhaps in the specification of a particular model.

2.2.4.2 Activities and System Instances

It could be possible to specify a relation between an activity and an instance of a system. The interpretation of such a relation might be that some activity is performed in the context of a particular system instance. For example, it may be required to perform system test (an activity) for some instance of a system. However, we felt that such a specification was more relevant to development of a model than the framework.

2.2.4.3 Events and Requirements

It did not appear appropriate to specify a relation between an event and a requirement. An example of how such a consideration may be of relevance is if the occurrence of some event could cause a change to a requirement. This consideration can be handled by building on the relation between events and activities and then composing them with operations that couple activities and requirements.

2.2.4.4 Events and System Instances

The reason that a relation was not declared that relates events and an instance of a system is similar to that described above in Section 2.2.4.2. Further, events are coupled to activities that can then be related to operations performed on an instance of a system.

2.2.4.5 Requirements and Requirements

The intended use of relations can be realized in different ways. That is, one can explicitly define a relation among requirements, or the same functional semantics can be achieved through the use of predicates. We chose to describe the relation among requirements using predicates to indicate, for example, that one requirement may depend on another requirement (see further Section 2.3.3).

2.2.4.6 System Instances and System Instances

Similar to the reasoning above, we have used predicates to indicate the relation between system instances. For example, we will later provide a mechanism to declare that one instance of a system depends on another instance (see further Section 2.3.4).

2.3 Predicates

The major building block for specification of operations in the language *Nestor* is a *predicate*. (Some background on predicates in *Nestor* can be found in Section B.4 on page 96.) Predicates can be represented in a number of ways including

- as a simple declaration: A simple example is a statement that a requirement has been tested. One can think of this as an axiom that is used to develop a specification.
- presence of a pre-condition clause: A predicate can include a specification of the conditions that must be satisfied in order for the predicate to be true. Such conditions are known as pre-conditions.
- presence of post-conditions: Given that a predicate has the value true, the post-conditions specify what happens under this circumstance. For example, one can say that if a requirement is not satisfied, then system testing is not completed.

While predicates are primarily used to specify operations associated with a specification, they can also be used as inquiry functions. In this case, one can declare a predicate that states that a requirement is satisfied. This fact (represented by the predicate) can then be used in the development of other predicates. A predicate may contain arguments; time is provided as a built-in capability.

2.3.1 Activities

Activities are a basic data type used in the specification of the acquisition framework. As such, it is useful to specify operations on the activities. Examples of this might include

- add an activity to the set of activities
- modify an existing activity
- remove an existing activity

Using the specification of an activity, we can write predicates for each of the above as follows:

Add_Activity (a_i: ACTIVITY; t, t') $\longrightarrow$ Boolean
$$\begin{aligned}&\models\!\!\Rightarrow\\ &\quad \text{Acq\_Activities}_{[t]} = \text{Acq\_Activities}_{[t']} \cup \{a_i\}\\ &\qquad t > t'\end{aligned} \tag{2.10}$$

In the above equation the parameters t and t' are present as arguments; these indicate values of time, which is a built-in data type in the language *Nestor*. In effect, Eq. (2.10) states that at the time t, the specified activity is added to the set of activities that existed at a time t', which must, of course, have been an earlier instant of time.

Modify_Activity (a_i: ACTIVITY; t) $\longrightarrow$ Boolean (2.11)

Delete_Activity (a_i: ACTIVITY; t, t') $\longrightarrow$ Boolean
$$\begin{aligned}&\models\!\!\Rightarrow\\ &\quad \text{Acq\_Activities}_{[t]} = \text{Acq\_Activities}_{[t']} \setminus \{a_i\}\\ &\qquad t > t'\end{aligned} \tag{2.12}$$

The first and third predicates above contain a specification of a post-condition. For example, if an activity is deleted, the result is that the specified activity should be removed from the set of acquisition activities. The second predicate has neither a pre- nor post-condition. It is interpreted simply as a declaration.

Note that the predicates declared above are associated with a Boolean value. This indicates the result of application of the predicate. For example, in the case where we can modify an activity, defined by Eq. (2.11), if a specified activity is modified, then the predicate assumes the value true. Hence, consider:

Modify_Activity (Requirements_Management; t) $\longrightarrow$ True

The result of the above operation is that it assumes the value true.

Each of the predicates specified above contains an argument (the symbol “t” appearing after the semi-colon) which is used in *Nestor* to denote time. We include time in the specification of a predicate in order to account for the dynamics of an acquisition. For example, we could use such an approach to state that “after a certain time no acquisition activity can be added to the set of acquisition activities.”

The ability to specify the model dynamics is fundamental to the specification of any acquisition.

There are a number of inquiry functions that may prove valuable for the development of a particular acquisition model. These are provided below.

Activity_Started (a_i: ACTIVITY; t) $\rightarrow$ Boolean (2.13)

Activity_in_Progress (a_i: ACTIVITY; t) $\rightarrow$ Boolean (2.14)

Activity_Suspended (a_i: ACTIVITY; t) $\rightarrow$ Boolean (2.15)

Activity_Completed (a_i: ACTIVITY; t) $\rightarrow$ Boolean (2.16)

The above predicates are true at a time t if an activity has been started, is in progress, has been suspended, or is complete, respectively. The inquiry functions can be used to construct other statements. For example, consider the following question: If an acquisition activity is in progress, can that activity be deleted? We would hope that the answer is no.[7, 8]

2.3.2 Events

In our basic declaration of events (see Section 2.1.2 on page 6), we provided for internal and external events. The difference between these two events was whether the event was initiated within or outside the scope of an acquisition project. There are a number of predicates relevant to a description of both these classes of events.

2.3.2.1 Internal Events

We begin by considering internal events. We define a predicate to create an event:

7. Incidentally, the answer to the question “if an acquisition activity is in progress, can it be deleted?” is no. The reason is that operations are assumed to be atomic in nature; that is, once they start to execute, they complete without interruption. See Section B.6 on page 99 for further discussion of this.

8. Some cases are fairly straightforward to develop, but there are subtle considerations one must account for. To digress a bit, consider the case where there is some external event. Assume further that the external event is related to some acquisition activity, as given by Eq. (2.8) on page 12. What behavior should an acquisition manifest if the activity associated with the event is deleted? In other words, if an external event arises (such as the existence of an upgrade to a COTS product that is used in a system) and there is no activity present to deal with the COTS product upgrade, what happens? Is the acquisition model specified correctly? It is through the use of not simply a formal mathematical *language*, but rather, the use of an approach that is based on formalism, logic, and completeness that leads one to develop (and consider) questions such as these.

Create_Internal_Event (e: INTERNAL_EVENT; t, t') $\rightarrow$ Boolean
$\models$
$\text{Internal\_Events}_{[t]} = \text{Internal\_Events}_{[t']} \cup \{e\}$
$t > t'$ (2.17)

If this predicate is true it means that the specified internal event has been created at the time t. It is then added to the set of internal events.

Another useful operation is to modify an existing event; we define this as

Modify_Internal_Event (e: INTERNAL_EVENT; t) $\rightarrow$ Boolean

A third operation is that which deletes some internal event; this is specified as

Delete_Internal_Event (e: INTERNAL_EVENT; t, t') $\rightarrow$ Boolean
$\models$
$\text{Internal\_Events}_{[t]} = \text{Internal\_Events}_{[t']} \setminus \{e\}$
$t > t'$

It is also helpful to specify a predicate that, if true, indicates that some internal event has occurred. This is a form of inquiry function, and we write it as

Internal_Event_Occurred (e : INTERNAL_EVENT; t) $\rightarrow$ Boolean (2.18)

We also need to be able to initiate, or raise, an internal event. We write this as

Generate_Internal_Event (e: INTERNAL_EVENT; t) $\rightarrow$ Boolean
$\models$ **Internal_Event_Occurred** (e : INTERNAL_EVENT; t)

In the preceding we have included a post-condition. That is, if some internal event is generated, it means that the predicate indicating that the event has occurred becomes true.

2.3.2.2 External Events

External events are those events that are specified in the context that lies outside of the scope of the acquisition project. At first one might think that we can simply replicate the operations associated with internal events, appropriately replaced instead by external events. However, such an approach is inappropriate. The reason is that from the perspective of the acquisition project, there are operations that can occur that are outside its scope. For example, the acquisition project cannot generate some external event. Instead, external events are generated by some (unspecified) agent that lies outside the scope of the acquisition project.[9] Including external events in the framework allows a coupling to items that are outside the scope of the acquisition project.

One operation that can be specified is that which indicates some external event has occurred. We write this as

$$
\begin{aligned}
&\textbf{External\_Event\_Occurred}\ (e_i\text{: EXTERNAL\_EVENT; } t) \rightarrow \text{Boolean} \\
&\quad |\!\rightarrow \\
&\qquad \exists\ a_j\text{: ACTIVITY} \bullet \\
&\qquad\quad \textbf{External\_Event\_Activity\_Relation}\ (e_i, a_j)
\end{aligned} \tag{2.19}
$$

Note the presence of a pre-condition here, in contrast to the corresponding operation for internal events (see Eq. (2.18)). That is, we are only interested in the occurrence of those external events such that they are related to an acquisition activity. In effect, the pre-condition clause serves to filter only the external events of interest to the acquisition project.

2.3.3 Requirements

Requirements management is a fundamental aspect of any acquisition, and the framework addresses requirements. One especially important consideration is that the ability to relate requirements to different instances of a system is of particular importance in acquisition (think of the difference between a waterfall model and an incremental mode, for example).

The predicates that will be included in the framework to perform operations can be developed from those for dealing with activities (see Eq. (2.10) ff.).

$$
\begin{aligned}
&\textbf{Add\_Requirement}\ (r_i\text{: REQUIREMENT; } t, t') \rightarrow \text{Boolean} \\
&\quad |\!\Rightarrow \\
&\qquad \text{Requirements}_{[t]} = \text{Requirements}_{[t']} \cup \{r_i\} \\
&\qquad\quad t > t'
\end{aligned} \tag{2.20}
$$

$$
\textbf{Modify\_Requirement}\ (r_i\text{: REQUIREMENT; } t) \rightarrow \text{Boolean} \tag{2.21}
$$

9. One could attempt to develop a specification of some relevant aspects of the external environment associated with some acquisition project. For example, one could develop a specification of a COTS marketplace. The integration of that specification and one based on the framework specified here would be achieved through the predicates defined here for dealing with such external events, as well as relations between the external events and the (internal) acquisition activities that are performed. Although the development of a specification for some external aspects of acquisition may be interesting and challenging, they are beyond the scope of the framework specified here. For some discussion of an acquisition model that includes COTS products, see [5].

$$
\begin{array}{l}
\textbf{Delete\_Requirement}\ (r_i\text{: REQUIREMENT; t, t'}) \rightarrow \text{Boolean} \\
\quad |\!\Rightarrow \\
\qquad \text{Requirements}_{[t]} = \text{Requirements}_{[t']} \setminus \{r_i\} \\
\qquad\quad t > t'
\end{array}
\tag{2.22}
$$

There are other operations that can be performed on requirements and warrant consideration as part of the framework. First, it is useful to know if some requirement has been satisfied. We can write a predicate for this as

$$\textbf{Requirement\_Satisfied}\ (r_i\text{: REQUIREMENT; t}) \rightarrow \text{Boolean} \tag{2.23}$$

A second operation deals with the specification of dependencies among requirements. We can define this by the following predicate:

$$\textbf{Requirement\_Dependency}\ (r_i, r_j\text{: REQUIREMENT; t}) \rightarrow \text{Boolean} \tag{2.24}$$

If the above predicate is true, it indicates that requirement r_i depends on requirement r_j at a time t. For example, this could mean that requirement r_j would have to be included in the system before requirement r_i. Using Eq. (2.24) it is also possible to develop requirements *chains*; that is, a sequence of dependencies. Thus, we could specify that requirement r_1 depends on requirement r_2, which depends on requirement r_3. This ability may be useful, in particular, to evolutionary acquisition models.

Given the ability to specify a dependency between requirements, we need to specify that there is not a pairwise dependency between requirements. In other words, if requirement x depends on requirement y, then we shall require that requirement y cannot depend on requirement x. Such a statement must hold for all requirements and is written as:

$$
\begin{array}{l}
\forall\, r_i \in \text{ Requirements} \bullet \\
\quad \textbf{Requirement\_Dependency}\ (r_i, r_j\text{: REQUIREMENT; t}) \\
\qquad \Rightarrow \\
\qquad\quad \neg\ \textbf{Requirement\_Dependency}\ (r_j, r_i\text{: REQUIREMENT; t})
\end{array}
\tag{2.25}
$$

The above is an *expression* in the language *Nestor*. It involves a quantification over all requirements and results in an assertion about the specification regarding requirements dependencies. As a practical application of this expression, suppose a specification of an acquisition model contained the following text:

Requirement_Dependency (Security_Requirements, Security_Requirements; t)
Requirement_Dependency (Security_Requirements, Security_Requirements; t)

By virtue of Eq. (2.25) the preceding declarations would be invalid.

2.3.4 System

We will define a predicate to indicate that some instance of a system depends on another instance of the system. This is written as

$$\textbf{System\_Dependency}\ (s_i, s_j\text{: SYSTEM; t}) \rightarrow \text{Boolean} \tag{2.26}$$

One of the important aspects of a system is in terms of how it relates to requirements. We define an operation that indicates that a particular requirement is satisfied by some instance of the system.

$$\textbf{Requirement\_Satisfied}\ (r_i\text{: REQUIREMENT, } s_j\text{: SYSTEM; t}) \rightarrow \text{Boolean} \tag{2.27}$$

If the above predicate is true it implies that the specified requirement is satisfied by a particular instance of the system. The framework does not specify the details of how a requirement is satisfied, as this is more a property of the system. However, we would expect that if a requirement has been satisfied, most likely it would mean that some testing has been done to verify the requirement.

Eq. (2.27) could be used to specify the time-phasing of requirements. For example, if we have two requirements denoted r_1 and r_2, then consider the following:

$$\begin{array}{l} \textbf{Requirement\_Satisfied}\ (r_1\text{: REQUIREMENT, } s_j\text{: SYSTEM; t}) \\ \textbf{Requirement\_Satisfied}\ (r_2\text{: REQUIREMENT, } s_j\text{: SYSTEM; t'}) \\ \quad t' > t \end{array}$$

The above specifies that requirement r_2 is satisfied at a later time than requirement r_1.

Finally, we can define a predicate that states that some instance of a system is satisfied. The pre-condition for such a predicate is that every requirement that is associated with a particular instance of a system must be satisfied. Mathematically, we have

$$\begin{array}{l} \textbf{System\_Instance\_Satisfied}\ (s_i\text{: SYSTEM; t}) \rightarrow \text{Boolean} \\ \quad \mapsto \\ \qquad \forall\ r_j \in \text{Requirements} \bullet \\ \qquad\quad \textbf{Requirements\_Mapping}\ (r_j, s_i) \bullet \\ \qquad\qquad \textbf{Requirement\_Satisfied}\ (r_j\text{; t}) \end{array} \tag{2.28}$$

In developing the preceding predicate we have used the relation between requirements and instances of a system, Eq. (2.9), and the predicate to indicate that a specified requirement has been satisfied, Eq. (2.23).

2.4 Execution Character

The specification developed thus far is largely static in nature. In reality, an acquisition has an execution characteristic, and we will now discuss this general topic. We will later develop several types of execution character that may be applied to a particular instance of the framework.

2.4.1 Basic Execution Model

In developing the predicates for the temporal evolution of an activity we will use a state model. We consider the following states of an activity: *null* (the initial state), *in_progress*, *suspended*, and *completed*. The state transition diagram appears in Figure 2-6.

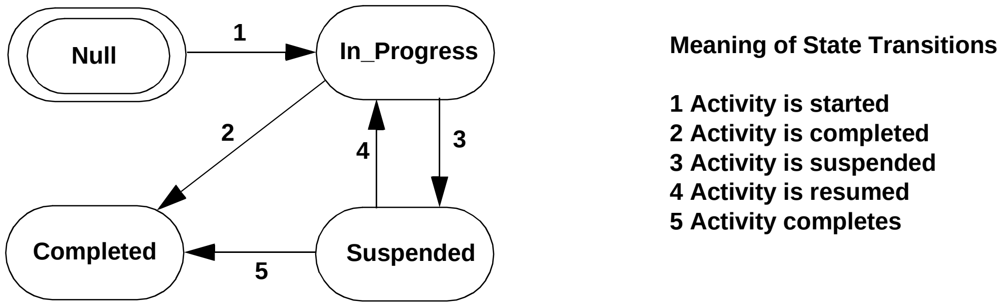


Figure 2-6: State Transition Diagram for an Activity

The state transition model shown in Figure 2-6 is simple, but it will serve our need for a specification in the framework. The first predicate we will declare is that which specifies the initiation of some acquisition activity. This is defined as

$$
\begin{aligned}
&\textbf{Initiate\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY; t}) \rightarrow \text{Boolean} \\
&\quad \mapsto \\
&\qquad \neg\ \textbf{Activity\_in\_Progress}\ (a_i\text{: ACTIVITY; t}) \\
&\quad |\!\Rightarrow \\
&\qquad \textbf{Activity\_in\_Progress}\ (a_i\text{: ACTIVITY; t})
\end{aligned}
\tag{2.29}
$$

The presence of the pre-condition states that an acquisition activity cannot be initiated unless it is not already in progress. The presence of a post-condition in Eq. (2.29) simply states that when an acquisition activity is initiated it is in progress.

We will also provide the capability to suspend and resume an acquisition activity. These are defined as:

$$
\begin{array}{l}
\textbf{Suspend\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY; } t) \longrightarrow \text{Boolean} \\
\quad |\!\!\rightarrow \\
\qquad \exists\, t' < t \bullet \textbf{Initiate\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY; } t') \\
\quad |\!\!\Rightarrow \\
\qquad \neg\ \textbf{Activity\_in\_Progress}\ (a_i\text{: ACTIVITY; } t)
\end{array}
\tag{2.30}
$$

Note that there is a pre-condition associated with suspending an acquisition activity; namely, the activity had to be started at a previous time. Note also the post-condition: if an activity is suspended, it is no longer in progress.

$$
\begin{array}{l}
\textbf{Resume\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY; } t) \longrightarrow \text{Boolean} \\
\quad |\!\!\rightarrow \\
\qquad \exists\, t' < t \bullet \textbf{Suspend\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY; } t') \\
\quad |\!\!\Rightarrow \\
\qquad \textbf{Activity\_in\_Progress}\ (a_i\text{: ACTIVITY; } t)
\end{array}
\tag{2.31}
$$

The predicate to resume an acquisition activity also has the pre-condition that the specified activity had to be suspended at an earlier instant in time. Also, the post-condition specifies that after an activity is resumed, it is now in progress.

Similarly, we define

$$
\begin{array}{l}
\textbf{Terminate\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY; } t) \longrightarrow \text{Boolean} \\
\quad |\!\!\rightarrow \\
\qquad \exists\, t' < t \bullet \textbf{Initiate\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY; } t') \\
\quad |\!\!\Rightarrow \\
\qquad \neg\ \textbf{Activity\_in\_Progress}\ (a_i\text{: ACTIVITY; } t)
\end{array}
\tag{2.32}
$$

As an aside, note that the specification of the predicate for suspending an acquisition activity is the same as the predicate for terminating an acquisition activity. When two predicates are identical we say that they are *underdefined*. This means that further specification, known as refinement, must be added to distinguish their semantics. In Section 5.8.3 on page 70 we will show how these predicates can be resolved by explicitly accounting for the state in their definition. Doing so removes the underdefined character of the specification.

We may also combine the predicates that define the initiation and termination of an activity to specify the execution of some acquisition activity in the following manner:

Execute_Acquisition_Activity (a_i: ACTIVITY; t, t') $\rightarrow$ Boolean
$\mapsto$
Initiate_Acquisition_Activity (a_i: ACTIVITY; t)
Terminate_Acquisition_Activity (a_i: ACTIVITY; t')
$t' > t$ (2.33)

The above predicate is defined in terms of two values of time t and t', which represent the initial and terminal times associated with the execution of the activity, respectively. The predicate **Execute_Acquisition_Activity** is true if and only if the predicates for the initiation and termination of the activity are also true. Note however, that it is possible for the predicate to be true even if the acquisition activity was suspended and resumed during the course of its execution.

2.4.2 Concurrency Considerations

A second facet to the execution of acquisition activities deals with the need to address concurrency considerations. It is a reality that acquisition frequently involves cases where multiple activities may be ongoing at the same time. In the framework it is thus relevant to specify operations related to concurrency.

The first operation that we will define is one that indicates that two different acquisition activities may be concurrent in nature. This is expressed as

Concurrency_Permitted (a_i, a_j: ACTIVITY) $\rightarrow$ Boolean (2.34)

The preceding predicate is assumed true if activities a_i and a_j are permitted to execute concurrently. One may also want to define an operation that specifies that two activities may not be concurrent, and we can write this as:

Concurrency_Prohibited (a_i, a_j: ACTIVITY) $\rightarrow$ Boolean (2.35)

It is also possible to develop an operation that is stronger in intent than that expressed by Eq. (2.34). That is, there may be occasions where it is required that certain activities execute concurrently; this can be expressed as

Concurrency_Required (a_i, a_j: ACTIVITY) $\rightarrow$ Boolean (2.36)

2.5 Completeness Considerations

It is important to assess whether an acquisition model is complete, and we will now develop an approach to this problem. Intuitively, it may appear clear what the notion of completeness means. However, one of the reasons for developing a formal specification is to be able to precisely define some aspect of a model. Hence, we will now formally describe what we believe is a reasonable definition of completeness of a general acquisition model. There are two aspects to completeness: of the specification and of an execution of that specification.

2.5.1 Specification Completeness

In general, the specification completeness indicates some property of the specification of an acquisition model as an instance of the framework. Several of such considerations have already been defined. For example, in the development of the relation between internal events and acquisition activities given in Eq. (2.7) on page 11 we required that every internal event must be related to some acquisition activity. This means that if an acquisition model declares an internal event, and that specified event is *not* related to some acquisition activity (through the specification of an instance of the relation), then the model is not complete.

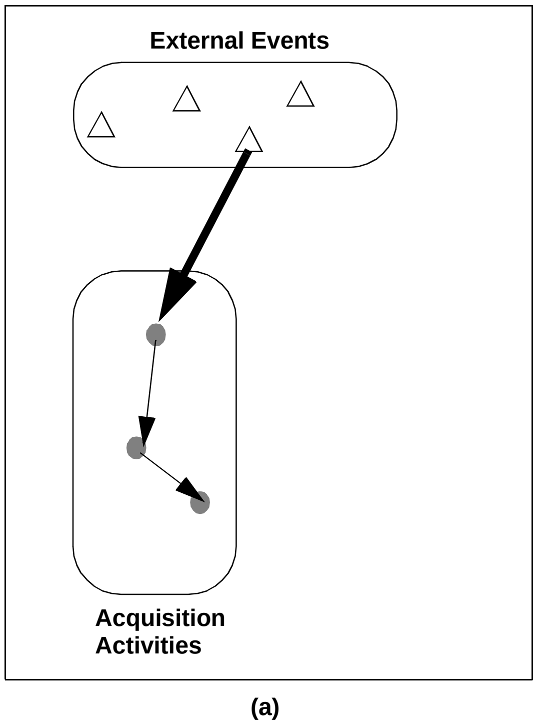
As another example, we required that the sets of internal and external events have to be disjoint, i.e., have no members in common. This was defined in Eq. (2.3) where we stipulated:

$$\text{Internal\_Events}_{[t]} \cap \text{External\_Events}_{[t]} = \varnothing$$

Hence, if a specification of a model had the same event in both sets, then the specification of the model would be incorrect.

To illustrate something that is not specified in the framework, we had considered adding a requirement that every acquisition activity had to be reachable from every other acquisition activity. This would mean there should be no “loose ends” in the way acquisition activities are related to each other. To assess the reachability criteria, consider the acquisition model “architectures” shown in Figure 2-7.

In part (a) of Figure 2-7 an external event is related to an acquisition activity, as required in the framework as described in connection with Eq. (2.8). However, if we required that each activity be related to some other acquisition activity, that is illustrated in connectivity of the activities shown in Figure 2-7(a). In contrast, consider part (b) of Figure 2-7. As required, there is still a relation between the external event and an acquisition activity. However, in this case, the indicated acquisition activity is not related to any other activity. Instead, there is a relation to an internal event, which then serves as an intermediary to other acquisition activities.



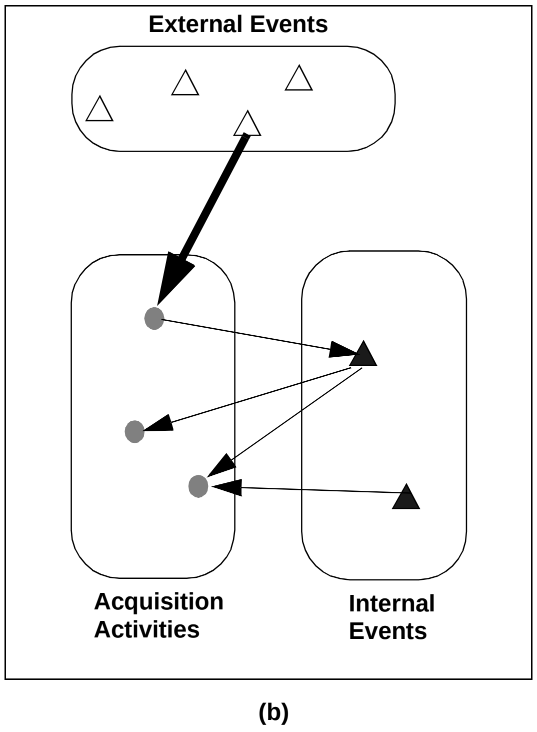


Figure 2-7: Different Mechanisms for Handling External Events

Hence, if we required a reachability property over the set of acquisition activities, it would preclude an architectural model such as that in part (b) of Figure 2-7. For this reason, we have not included a reachability property in the framework. An acquisition model may, however, impose such a condition without adverse effect on the framework specification. For a further elaboration of this point, see the discussion dealing with conformance of a model to the framework in Section C.1.

2.5.2 Execution Completeness

There is a second view of completeness of a model, and that is with respect to the execution of the model. From a practical perspective, we are talking about the manner in which the model is performed.

We begin by declaring two instants of time that denote an interval over which an acquisition takes place. These may be interpreted as the "beginning" and "end" of the acquisition. The declaration for these instants is

t_{start}, t_{stop}: Time

From an operational view, we state that an acquisition model is execution-complete on the interval $[t_{start}, t_{stop}]$ if the following conditions are satisfied:

- The execution of acquisition activities is well specified on the interval. By this we mean that no activity starts before the time t_{start}, and that all activities must be terminated prior to the time t_{stop}.
- For every external event that may be initiated in the interval and that is related to some acquisition activity, that acquisition activity must be initiated before the end of the interval. The preceding item further implies that the activity associated with the external event must also complete.
- For every internal event that may be initiated in the interval, the event is related to some acquisition activity, and its associated acquisition activity is initiated.
- For every requirement that exists at the end of interval t_{stop}, the requirement is satisfied.

Well Specified

The criterion of well specified may be stated as

$$\begin{aligned}
&\forall\, a_i \in \text{Acq\_Activities}_{[t]} \\
&\quad \exists\, t_1 \geq t_{start} \bullet \textbf{Initiate\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY}; t_1) \land \\
&\qquad \exists\, t_2 \leq t_{stop} \bullet \textbf{Terminate\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY}; t_2) \\
&\qquad\quad t_{start} < t_{stop}
\end{aligned} \tag{2.37}$$

Note that the preceding does not mean that an activity cannot be suspended and then resumed (perhaps multiple times) during the interval $[t_{start}, t_{stop}]$. The key point is that the activity be terminated before the end of the time interval in consideration.

External Events

The next condition for execution completeness is that for those external events that may occur (as defined by the relation between external events and acquisition activities) during the interval, the associated acquisition activity must be initiated.[10] The necessary formalization of this condition is as follows:

10. There is one potential loose end, namely, if an external event occurs before the start time of the interval, but the acquisition activity that deals with the event does not start until after the beginning of the interval; hence, a possible overlap. However, as we shall see, if a model is execution complete on an interval, then it is not possible for the above situation to occur on any other than the first interval.

$$\begin{array}{l} \forall\ e_i \in \text{External\_Events}_{[t]} \bullet \\ \quad \textbf{Time\_of [External\_Event\_Occurred}\ (e_i\text{: EXTERNAL\_EVENT; t)}\textbf{]} \in [t_{start}, t_{stop}] \\ \quad\quad \exists\ a_j\text{: ACTIVITY} \bullet \\ \quad\quad\quad \textbf{External\_Event\_Activity\_Relation}\ (e_{i\ ,}\ a_j) \\ \quad\quad \wedge \\ \quad\quad\quad \textbf{Initiate\_Acquisition\_Activity}\ (a_j\text{: ACTIVITY; t)} \\ \quad\quad\quad\quad t \in [t_{start}, t_{stop}] \end{array} \tag{2.38}$$

Note the presence of **Time_of** in Eq. (2.38). This is a built-in function in *Nestor* that extracts the time associated with the evaluation of its argument. In this case, the operation **Time_of** returns the value of time t associated with the operation **External_Event_Occurred**; in other words, it returns the time when the external event occurred.

We make an important note concerning Eq. (2.38). From Eq. (2.37) we see that if an acquisition activity is started on the interval, it must be completed. The specification of Eq. (2.38) only states that the acquisition activity is started. However, it follows from these two equations that the activity associated with the external event must also terminate. This is an example of reasoning over a specification, which is an important consideration for more advanced work.

Internal Events

The next condition is that associated with internal events. This condition may be specified in a manner similar to that for external events, described above. We have

$$\begin{array}{l} \forall\ e_i \in \text{Internal\_Events}_{[t]} \bullet \\ \quad \textbf{Time\_of [Internal\_Event\_Occurred}\ (e_i\text{: INTERNAL\_EVENT; t)}\textbf{]} \in [t_{start}, t_{stop}] \\ \quad\quad \exists\ a_j\text{: ACTIVITY} \bullet \\ \quad\quad\quad \textbf{Internal\_Event\_Activity\_Relation}\ (e_{i\ ,}\ a_j) \\ \quad\quad \wedge \\ \quad\quad\quad \textbf{Initiate\_Acquisition\_Activity}\ (a_j\text{: ACTIVITY; t')} \\ \quad\quad\quad\quad t \in [t_{start}, t_{stop}] \end{array} \tag{2.39}$$

The interpretation of Eq. (2.39) is easily discerned from that described above for external events.

Requirements

Our final condition for execution completeness is that all requirements that exist at the end of the interval must be satisfied. This condition may be specified as follows:

$$\begin{array}{l} \forall\ r_i \in \text{Requirements}_{[tstop]} \bullet \\ \quad \textbf{Requirement\_Satisfied}\ (r_i;\ t_{stop}) \end{array} \tag{2.40}$$

Note the power achieved through the use of the dynamic data type used to represent requirements. In particular, $\text{Requirements}_{[tstop]}$ denotes the members of the set of requirements at the time t_{stop}. We then simply require that each of these requirements must be satisfied. Note further, if a requirement is deleted before t_{stop} then it will not be in the set $\text{Requirements}_{[tstop]}$ and we need not be concerned with it. Furthermore, if some requirement is added prior to t_{stop} it will remain in the set (unless otherwise deleted) and must therefore also be satisfied. Hence, we do not care what changes are made to the set of requirements, either through addition, deletion, or modification. We simply require that all the requirements at the end of the interval must be satisfied.

Summary

The preceding has described an approach that allows us to specify that an execution of some acquisition model is complete with respect to some time interval. We have developed this approach in terms of activities, internal and external events, and satisfaction of requirements.

We have not addressed the seemingly innocuous question of when an execution of a model is entirely complete; i.e., when is it done? There are different ways to define overall completeness, including when

- all of the requirements are satisfied
- all activities have terminated
- the latter of the above completes

Any of the above are reasonable definitions of when an execution of a model completes. Because each is acceptable, we do not include, in the framework, a definition of when an execution of a model is done. Instead, that choice is left to the developers of a particular model specification.

We conclude by noting a practical benefit to the concept of execution completeness. For example, consider an acquisition model that is based on an incremental or spiral approach. When is an increment or spiral complete? It is the use of a specification such as presented above that allows us to address this concern. When recast in the context of an acquisition program, performed according to a particular acquisition model, then the question becomes very relevant in a practical sense.

2.6 Summary

By means of summary, and largely from an intuitive perspective, the following figure summarizes the relations between acquisition activities, internal and external acquisition events, and requirements and instances of a system.

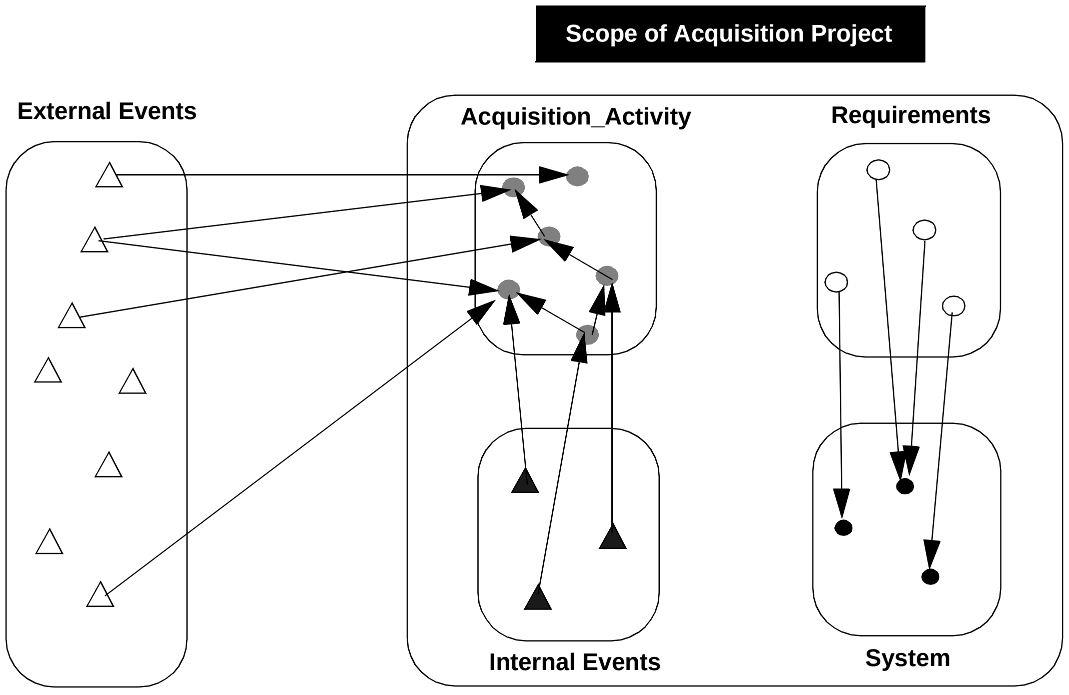


Figure 2-8: Intuitive Relation of Overall Acquisition Model Elements

The relations developed in Section 2.2 on page 9 are indicated in the above figure. For example, there are relations between external events and acquisition activities. Note also the relation among different acquisition activities is illustrated in Figure 2-8. The structure presented in Figure 2-8 is static in nature, and does not show the dynamics associated with the framework.

A summary of the framework from a mathematical perspective may be found in Appendix A on page 77. There we group together the various mathematical aspects of the framework.

We should reiterate our interest in developing the framework specification and the question of scope; that is, what amount of detail does one include in the framework, as opposed to a model based on that framework? Later, in Section 5 on page 49, we will address possible extensions that could be incorporated in the framework. However, at this point we believe the framework satisfies the proper balance in terms of its ability to express (and later refine) acquisition concepts and the need for additional detail that is required for treatment of a particular acquisition model based on the framework.

3 An Example: The Waterfall Model

To provide an illustration of the application of this formal approach, we will consider how a (pure) waterfall development model can be specified. Originally specified by Royce [6], this model was a mainstay of systems development for many years. It has fallen out of favor in recent years, yet is still applied in the context of other, more recently developed, acquisition models. In the following we will develop a specification of the waterfall model in the context of the acquisition framework. Some extensions to the waterfall model specification, based on extensions to the framework, are described in Appendix D.

3.1 Basic Specification Elements

Activities

The waterfall model [6] is characterized by a set of activities. Its simplest representation is shown in Figure 3-1.

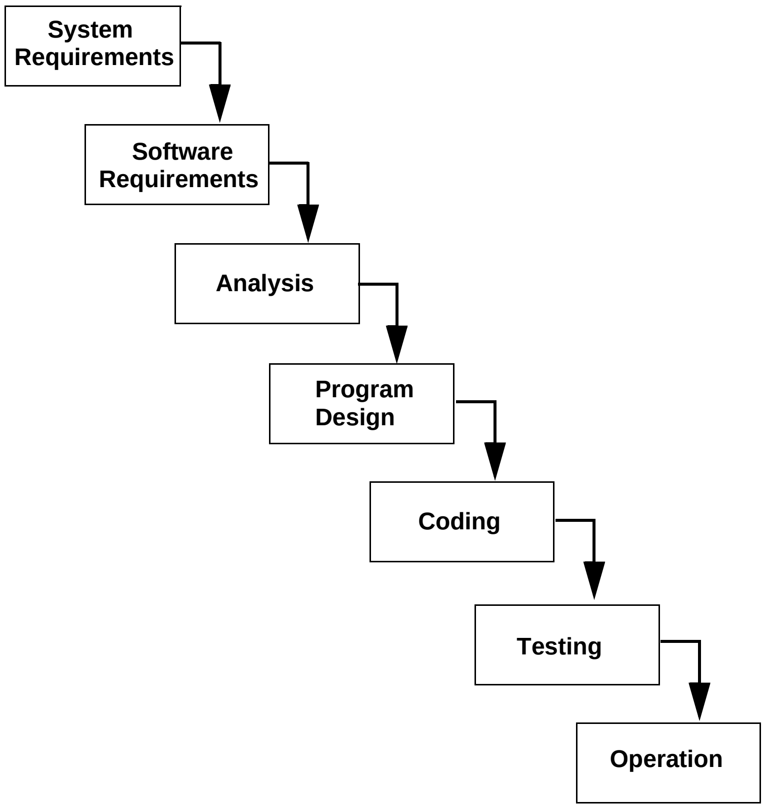


Figure 3-1: Basic Representation of Waterfall Model

The choice of presentation in Figure 3-1 is meant to indicate, in its simplest case, that the activities are performed in the order shown; hence the term waterfall. The arrows shown in Figure 3-1 represent relations between the indicated activities.

The set of activities shown in Figure 3-1 may be represented in the framework as follows:

Acq_Activities: {System_Requirements_Definition, Software_Requirements_Definition, Analysis, Program_Design, Coding, Testing, Operation} (3.1)

Note that we do not include a subscript [t] on the above. The lack of the subscript [t] means that the data object is static, which is the case for the activities defined for the waterfall model.

Events

The second aspect of the framework is the specification of events. The first group of events are those that are internal to the scope of the project. For the waterfall model, there are items described that we will interpret as events. These are reviews defined in the context of interaction with the customer, and we can declare them as

Preliminary_Software_Review, Critical_Software_Review, Final_Software_Acceptance_Review: INTERNAL_EVENT

Internal_Events: {Preliminary_Software_Review, Critical_Software_Review, Final_Software_Acceptance_Review} (3.2)

The second group of events are those that are external to the scope of the project. In the case of the pure waterfall model there are no such external events; this constraint is specified by

$\#\text{External\_Events} = \varnothing$

The fact that there are no external events indicates the development does not involve external factors, such as the use of COTS products.

Requirements

In terms of requirements, we introduce the following:

```
[REQUIREMENT]

Requirements: {REQUIREMENT}
NUMBER_REQUIREMENTS: Number
#Requirements > 0
#Requirements = NUMBER_REQUIREMENTS
```

The preceding specifies a finite set of requirements whose number is defined by the constant `NUMBER_REQUIREMENTS`. It is important to note that the number of requirements is assumed constant. This is in keeping with the assumption of the waterfall model, in that there is a fixed set of requirements that are then used to develop the system. This further implies that, in the development of operations that might be performed on requirements, that one cannot either add or delete a requirement. Hence, such operations are not necessary in the waterfall model, although they remain useful in the context of a framework.

System

We specify the system as follows:

```
[SYSTEM]
System: {SYSTEM}
#System = 1
```

The above declares that there is only one instance of the system. In recognition that the system needs to be maintained, we could specify a set of system instances where a given instance of a system is determined by the changes that it incorporates during maintenance. We shall not take this approach however, because it does not lend anything of interest to the model. Furthermore, recall that the emphasis on the specification of the waterfall model in [6] was development, and we continue that emphasis here.

Relations Among Activities

As for the relations among the activities, based on Figure 3-1 one might assume that the relations should be defined for only "nearest neighbor" activities. In other words, there would be a relation defined between *Program Design* and *Coding,* but not a relation between *Software Requirements Definition* and *Coding.* However, the representation shown in Figure 3-1 is the currently popular rendition of the waterfall model and does not account for its original specification. In particular, the original specification admitted not only "nearest neighbor" relations, but recognized the necessity for other relations among activities. The basic specification of a relation among activities applies here, that is

Activity_Relation: Activities <-----> Activities
$\exists$ a_i, $a_j \in$ Activities
• **Activity_Relation** $(a_{i\,,}\, a_j)$
$a_i \neq a_j$ (3.3)

The preceding specifies that every activity is related to every other activity except itself. Another way to represent relations, whether that defined above or in general, is to use a matrix representation of the relation. In this case, we would have

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Figure 3-2: Matrix Representation of Activity Relations in the Waterfall Model

It is assumed that the rows and columns in Figure 3-2 are ordered in correspondence with the activities declared in Eq. (3.1). The value unity denotes a relation exists between the indicated activities, and the value zero means there is no relation specified.

Relations Among Events and Activities

The second type of relation in the framework is that between events and activities. In terms of reference [6] the relation can be declared as

Project_Review: Internal_Events ----> Acq_Activities
$\exists$ $e_i \in$ Internal_Events, $a_j \in$ Acq_Activities
• **Project_Review** $(e_{i\,,}\, a_j)$ (3.4)

The character of the relation in Eq. (3.4) is one-to-one: each internal event must relate to some activity, but not all activities must be related to an event.

The internal events declared in Eq. (3.2) are related to certain activities as discussed in [6]. In particular, the following relations for project reviews are described:

Project_Review (Preliminary_Software_Review, Preliminary_Program_Design) (3.5)

Project_Review (Critical_Software_Review, Coding) (3.6)

Project_Review (Final_Software_Acceptance_Review) (3.7)

For example, the first relation indicates a relation between an internal event (the preliminary software review) and its related activity (preliminary program design).

Because there are no external events associated with the waterfall model, there are no such relations that we need to consider.

3.2 Predicates

In Section 2.1.1 on page 5 we defined predicates for activities that are part of the framework. For example, we provided predicates to add, modify, or delete an acquisition activity. In the case of the waterfall model, such predicates are not necessary. The reason is that the activities are pre-defined, hence there is no need to define operations to change them.

For internal events, the framework specification accounts for predicates that allow us to add or modify an internal event. Again, because the internal events are pre-defined in the waterfall model, it is not necessary to define predicates that perform such operations. However, two predicates that may be useful in the specification of the waterfall model are the following:

Internal_Event_Occurred (e : INTERNAL_EVENT; t) $\rightarrow$ Boolean

Generate_Internal_Event (e: INTERNAL_EVENT; t) $\rightarrow$ Boolean
$\quad\mid\Rightarrow$ **Internal_Event_Occurred** (e : INTERNAL_EVENT; t) (3.8)

For example, using the above predicates it may be possible to indicate that when some activity completes, a certain internal event is started.

Because the waterfall model does not include external events, it is not necessary to define predicates that deal with external events.

For requirements it will be useful to carry over the following predicates from the framework specification:

Modify_Requirement (r_i: REQUIREMENT; t) $\longrightarrow$ Boolean (3.9)

Requirement_Satisfied (r_i: REQUIREMENT; t) $\longrightarrow$ Boolean (3.10)

Although it is assumed that the requirements on the system are known at the outset of the project, we need to include a predicate that allows us to modify a requirement. It was recognized that there may be a need to go back and modify an initial requirement. However, we have not included predicates to create or delete a requirement because their number is assumed fixed.

Finally, for the specification of predicates related to the system, since there is only one instance of the system, we need not define predicates that relate to various system considerations.

The preceding discussion of predicates in the context of the waterfall model has turned out to be quite simple and straightforward. The reason for this is because so many aspects of the model are based on inherently simple considerations. For example, a fixed set of requirements and only one instance of the system is quite easy to deal with.

3.3 Timing Properties

To begin a discussion of the timing properties associated with the waterfall model, we define the times when an implementation of the model starts and stops:

t_{start}, t_{stop}: Time

We begin a specification of the timing properties of the waterfall model by carrying over the definitions to initiate and complete an acquisition activity.

Initiate_Acquisition_Activity (a_i: ACTIVITY; t) $\longrightarrow$ Boolean
  $\mid\!\!\longrightarrow$
    $t \geq t_{start}$ (3.11)

Terminate_Acquisition_Activity (a_i: ACTIVITY; t) $\longrightarrow$ Boolean
  $\mid\!\!\longrightarrow$
    $\exists\, t' < t \bullet$ **Initiate_Acquisition_Activity** (a_i: ACTIVITY; t')
      $t' \geq t_{start}$
      $t \leq t_{stop}$ (3.12)

Note the pre-conditions placed on the above equations. The operation to initiate an acquisition activity requires that the time at which the activity is initiated must be later than the time when the project started. Similarly, the time at which an activity terminates must be earlier than the time at which the project completes.

To specify the initiation of a waterfall development, we require that the first activity performed is that associated with system requirements. In other words, at a time equal to t_{start} the system requirements activity must be initiated. Mathematically, this can be specified as:

Initiate_Acquisition_Activity (System_Requirements_Definition; t_{start}) $\rightarrow$ Boolean

Given that the model implementation is initiated, the permitted activities that may later be formed are those activities that are related to the activity just completed. It is through the coupling of activities, specified by the relation defined in Eq. (3.3), that defines the activities that can be performed. This requirement can be specified as follows:

$\forall\, t \in (t_{start}, t_{stop}) \bullet$
  Terminate_Acquisition_Activity (a_i: ACTIVITY; t)
$\Rightarrow$
  Initiate_Acquisition_Activity (a_j: ACTIVITY; t)
    $\bullet$ **Activity_Relation** (a_i, a_j)
      $a_i \neq a_j$ (3.13)

Hence, when some activity terminates, the next activity that may be performed is one such that it is related to the activity that completed. This is a general form of constraint on the execution characteristics associated with a model. In the case of the waterfall model, however, once an activity terminates, any other activity can be performed. The reason for this is because the relation between acquisition activities is that every activity is related to every other activity (except itself). Although most diagrams representing the waterfall model only show activities coupled in a nearest-neighbor fashion, as shown in Figure 3-1, the specification of the model in [6] is more general.

An interesting question is: When does the implementation of the waterfall model end? On the one hand, one could say “never” because a system goes into operation and maintenance after it is developed! In the definition of activities for the waterfall model, the last two activities are associated with testing and operation. We will assume that the implementation of the model ends when both of the following two requirements are met:

- all requirements are satisfied
- the activity of operation begins

The framework includes an operation that specifies that a given requirement is satisfied (see Eq. (2.23) on page 21). To specify that *all* requirements are satisfied we promote the operation that a given requirement is satisfied over the set that includes all the requirements. We can introduce an expression for this case as follows:

```
**All_Requirements_Satisfied** (t*) → Boolean
    |→
        ∃ t* •
            ∀ $r_i$ ∈ Requirements •
                **Requirement_Satisfied** ($r_i$: REQUIREMENT; t)
                    t ≤ t*                                              (3.14)
```

In other words, the above states that there is a time t* when every requirement has been satisfied.

We may now use the result in Eq. (3.14) to specify the condition that an implementation of the waterfall model is complete. We have:

```
**Waterfall_Model_Complete** (t*) → Boolean
    |→
        **All_Requirements_Satisfied** (t*)
        **Initiate_Acquisition_Activity** (Operation; t*)
            $t^* \le t_{stop}$                                          (3.15)
```

As indicated earlier, other interpretations of what it means for an implementation of a model to be complete can be provided. It could be the first time that all requirements associated with the system are satisfied, as was the choice represented in Eq. (3.15). One could refine the model to account for deployment of a system and assume that the time of deployment is when the model is complete. Or, yet another choice would be to define the completion time as when the system is retired.

From the preceding we have specified the timing properties associated with the waterfall model. An interesting question is to address the sequence of activities that are performed as part of a waterfall acquisition. We will not provide the mathematics of this (although it is simple to do so). Instead, in Table 3-1 we show three possible sequences of activities for a waterfall model.[11]

The first column of Table 3-1 represents the basic (ideal) model presented in Figure 3-1. The second column shows activities that include the iteration between some of the nearest-neighbor activities. For example, after performing the design, coding, and test, it may be necessary to go back and perform some coding activity. The last column of Table 3-1 shows a more general case where activities may be performed in an arbitrary order. After performing the activities in the basic approach, after testing it may be necessary to go back to the software requirements definition and then start the sequence over again.

11. A specification of a sequence of activities is known as a *trace*.

	Basic	Nearest Neighbor	General
1	**System Requirements Definition**	**System Requirements Definition**	**System Requirements Definition**
2	**Software Requirements Definition**	**Software Requirements Definition**	**Software Requirements Definition**
3	**Analysis**	**Analysis**	**Analysis**
4	**Program Design**	**Program Design**	**Program Design**
5	**Coding**	**Analysis**	**Coding**
6	**Testing**	**Program Design**	**Testing**
7	**Operation**	**Coding**	**Software Requirements Definition**
8		**Testing**	**Analysis**
9		**Coding**	**Program Design**
10			**Coding**

Table 3-1: Valid Activity Sequences in the Waterfall Model

Finally, we consider the timing properties of internal events. By defining the relation between an internal event and some activity as we did in Eq. (3.4), we know which events are associated with a given activity. We now require that the internal event occurs at some time during the execution of its associated activity. This can be specified as

$$\begin{array}{l} \forall\ a_i \in \text{Acq\_Activities} \bullet \\ \quad \textbf{Project\_Review}\ (e_i,\ a_j) \\ \qquad \exists\ t^* \in \textbf{Duration}\ \{\textbf{Execute\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY; t, t'})\} \bullet \\ \qquad\quad \textbf{Internal\_Event\_Occurred}\ (e_i,\ a_j) \end{array}$$

The above simply states that for every activity that has an associated event, the event occurs at some time during the execution of that activity. No further requirement is placed on the actual time at which the event occurs during the activity.[12]

[12.] The function of **Duration** is a projection operator that extracts the times associated with the predicate on which it operates. This is a built-in function provided by the language *Nestor*.

3.4 Completeness Considerations

The specification of the framework contains an important discussion about the completeness criteria associated with a particular model. A brief summary of the completeness criteria and an index to where they are discussed for the waterfall model is provided in Table 3-2.

Topic	Section
Execution of activities is well specified	3.4.1
Treatment of external events	3.4.2
Treatment of internal events	3.4.3
All requirements satisfied	3.4.4

Table 3-2: Completeness Considerations for Waterfall Model

3.4.1 Execution of Activities Is Well Specified

The criterion of well specified means that no activity begins at a time before the model starts, and no activity ends after the model completes. This is described in connection with Eq. (2.37) on page 28. The criteria for well-specified for the waterfall model are expressed as the pre-conditions associated with Eqs. (3.11) and (3.12).

3.4.2 Treatment of External Events

It is easy to handle this criterion for the waterfall model: The model does not admit external events. Hence, this requirement does not apply.

3.4.3 Treatment of Internal Events

The waterfall model only specified three internal events, namely preliminary software review, critical software review, and final software acceptance review. The completeness requirement for internal events, as expressed by Eq. (2.37) on page 28, is that every internal event must be related to some acquisition activity. This criterion is satisfied by Eq. (3.4) and following.

3.4.4 All Requirements Satisfied

The last completeness criterion is that all requirements must be satisfied. This was specified in Eq. (3.14) on page 40. Furthermore, the specification that all requirements were satisfied was used as part of the pre-condition to the specification that the model is complete.

3.5 Reasoning About the Specification

One of the advantages of a formal, mathematical approach is that it lends itself to reasoning in a precise manner. This is typically described under the ability to state, and complete, proofs about some aspect of a specification. Our intent here is not to go into details about application of proof-theoretic techniques about the specification of the waterfall model. However, let us consider some small examples that deal with reasoning about the specification.

Can the same activity be executed immediately after it has terminated? No. The reason is evident from Eq. (3.13) and the restriction that $a_i \neq a_j$. An activity can be initiated only if it is related to another activity, and an activity is not defined as relating to itself.

Must every activity be executed at most one time? No. The specification of the waterfall model does not make any such restriction. An implementation of the specification of the waterfall model may, however, exhibit such behavior.

If an implementation of the model completes, what happens if a requirement is later changed? The specification is silent on this point. That is, there is an assumption that all the requirements are known at the start of the model and thereafter constant. There are not operations in the specificaiton of the waterfall model to add or delete a requirement, although there is an operation that allows a requirement to be modified.

What happens if an acquisition activity is suspended; can there be times when no activity is in progress? Again, the specification is silent on this point. While the general specification of the framework does provide operations for an activity to suspend and resume, those operations are not incorporated in the specification of the waterfall model. The reason is that the specification is silent with respect to this question.

The above illustrates simple cases where one can pose a question of the specification and obtain a response that can be substantiated mathematically. It is this point—the ability to formally reason over the domain of a specification—that is so valuable to a formal approach.

3.6 Summary

There are a number of points that should be noted as a summary of a formal specification of a waterfall model. Two key points include the following:

- The formal specification serves to validate the utility of the framework. The specification of activities, events, and their relations, as well as execution character, has allowed a precise specification of the waterfall model. Note that not all of the framework capabilities were required to specify the waterfall model.

- The development of the formal specification has required us to carefully define the model with the precision required of a mathematical approach. This is one of the well-known benefits of a formal approach.

We believe that the formal approach taken here can also be applied to other models that are used for an acquisition. Some of these will be noted in the following section. If one can develop a formal specification of an acquisition model based on the framework developed in Section 2, with possible extensions, it allows one to compare the different acquisition models in a more enlightened manner. This, in turn, helps one gain deeper understanding of an acquisition model.

4 Discussion

The framework that has been developed in this report is purposefully general in nature. The intent is that the framework can serve as the basis for the development of particular acquisition models, which are instances of this framework. This approach is indicated in the following figure:

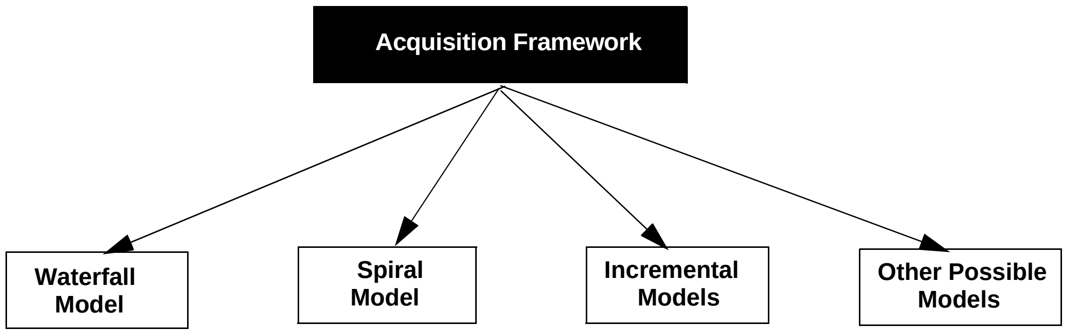


Figure 4-1: Acquisition Models

Figure 4-1 shows some possible acquisition models that could be specified in terms of the acquisition framework. We recognize and expect that it may be necessary to add additional details to the basic framework specification as one applies the framework to a particular model. The models indicated here are:

- waterfall model: the well-known model based on the work of Royce [6], which was illustrated in Section 3
- spiral model: another well-known model, based on the work of Boehm [1]
- incremental models, also sometimes considered to be evolutionary in nature
- other possible acquisition models

Although the above represent some of the well-known models that are discussed today, they can be embellished in different ways. For example, if one seeks to perform an acquisition that places heavy emphasis on the use of open systems and COTS products, this introduces additional considerations that must be accounted for in the model; see [5] for further discussion of this point.

A simplified comparison of the above models, in terms of concepts employed in the framework developed here, is summarized in Table 4-1.

Framework Element	Waterfall	Spiral	Incremental
Requirements	Assumed known at start of project	Developed as the project progresses with each iteration of the spiral	Assumed known at start of project
Activities	System Requirements, Software Requirements, Analysis, Program Design, Coding, Testing, Operation	Top level specification includes planning, risk management, engineering, and customer evaluation	Various choices possible; e.g., those associated with the waterfall model
Events	Preliminary software review, critical software review, final software acceptance review	Various choices possible	Various choices possible
Relations	Arbitrary coupling between activities permitted, but mainly expected to be between nearest neighbors	Coupling between each activity in a cycle and end of one cycle to next cycle	Various, based on activities chosen
Timing Properties	Activities performed serially	Activities performed serially within repetitive cycles	Activities performed serially; each increment adds more functionality (i.e., satisfied requirements)
System Instances	Focus on only one instance of the system	Multiple instances determined during execution of acquisition model	Pre-planned multiple system instances

Table 4-1: Comparing Different Acquisition Models

We emphasize the simplified view presented in the above table. It is more illustrative of the characteristics of these models, as opposed to details associated with them. No doubt proponents of one model or another may cry foul at the simplification. In response we would welcome well-specified definitions of these models in the context of the framework defined here.

One of the primary differentiators of the models shown in Table 4-1 is the manner in which the requirements are satisfied by an instance of a system over time. In the waterfall model, for example, all requirements are assumed known at the outset, and one proceeds to develop the (hopefully) final instance of the system. In contrast, a spiral development approach adds requirements over time that are implemented in different instances of the system over time, leading to a desired final instance of the system.

Although the approach to satisfying requirements as part of a system acquisition is a differentiating characteristic, there are others that deserve attention. We would suggest that the increased emphasis on how open standards and COTS products are addressed in an acquisition is another key characteristic of an acquisition model. In particular, consideration of standards and their relation to COTS products leads one to deal with models where external events (such as the upgrade of a COTS product) are crucial to the acquisition. Further discussion of open standards, COTS-based acquisition may be found in [5].

Each of the above models is comprised of different activities, events, and relations among the activities, including their operational semantics. We believe there would be significant value in the formal specification of the above models. Being able to compare the models in a formal context adds value not only in understanding the model itself, but how one model relates to other models.[13] Furthermore, the development of a robust specification could be used to help manage an acquisition project, and we would expect a work breakdown structure could be *derived* from a formal model.

It is out of the scope of this report to develop a formal specification for all the models listed above. We would hope, however, that this framework can be applied to various acquisition models, which are often only loosely stated. The existence of a formally specified acquisition model, such as a spiral model, for example, would allow one to then speak of a *theory of spiral acquisition*. It is through the application of a formal approach that we are allowed not only to specify but to *reason about* various acquisition models.

13. To illustrate some of the confusion that exists, we recently reviewed a document consisting of over 3,000 supposedly completed requirements for a system. That was fine, until the claim was made that this was going to be a spiral acquisition!

5 Possible Extensions to the Framework

Our goal in specifying the framework in Section 2 was to take a minimalist approach. That is, we wanted to limit the scope of the specification with respect to the amount of detail present. There are other possible topics that could be included in the framework, although we have chosen not to take that approach. One of the basic problems in specifying any type of framework is to know when to stop! Clearly, there are other topics that could be relevant for a model specification based on the framework defined here. In this section we will illustrate some other possible considerations.

5.1 Elaboration of Acquisition Activities and Events

The framework does not specify any particular set of acquisition activities. The reason is that the choice of activities is deemed relevant to a particular model based on the framework. Although we could have attempted to declare a particular set of activities, we realize there is always the case where model developers will require perhaps different activities.

Furthermore, we have not explicitly specified that an acquisition activity could be divided into a set of other activities. For example, consider an activity *Code Construction*. We could envision that this activity be further refined to include more detailed activities such as *Code Development, Code Inspection, Code Guideline Development,* and so on. There is nothing in the framework that prevents an activity from being defined as a *subset* of other activities. If this approach is required, it can be easily incorporated in a particular model.

In a similar manner, if deemed necessary, one could partition the acquisition activities into two subsets. One subset would be used to represent technical activities, while the other would be for management activities. We would then define

$$\text{Technical\_Activities}_{[t]}\text{: \{ACTIVITY\}}$$
$$\text{Management\_Activities}_{[t]}\text{: \{ACTIVITY\}}$$
$$\text{Technical\_Activities}_{[t]} \cap \text{Management\_Activities}_{[t]} = \varnothing$$

The last line requires that the sets of technical and management activities have no activity in common.

One reason for treating management and technical activities separately is to recognize their separate importance. Taking this view allows for a "divide and conquer" approach. Additionally, we note that the technical activities are typically of bounded duration, while the management activities are performed over the entirety of a project life cycle.

The preceding discussion also applies to the events that are included in the framework. That is, we do not feel it is appropriate to attempt to specify each possible event that could be included in the framework.

We do believe, however, that the separation of events into internal and external events is fundamental and does belong in the framework (as well as the relations between events and activities). One of the characteristics of current acquisition practice is the need to deal with standards and commercial off-the-shelf (COTS) products. There are events associated with each of these activities; for example, the revision to some COTS product would be represented as an external event. Further discussion of standards-based COTS acquisition can be found in [5].

5.2 Requirements

The framework postulates a finite set of requirements and specifies that those requirements are related to one (or possibly more) instances of a system. There are a number of operations that can be performed on requirements that are not part of the core framework specification. For example, one could define the following operations:

Requirement_Correct (r_i: REQUIREMENT; t) $\rightarrow$ Boolean

Requirement_Self_Consistent (r_i: REQUIREMENT; t) $\rightarrow$ Boolean

Requirements_Consistent (r_i, r_j: REQUIREMENT; t) $\rightarrow$ Boolean

We did not include the above predicates in the core framework specification because such predicates are more focused toward a specification of an acquisition model. Recall that the main concern in specifying the framework is to include those operations and data structures that are useful in the characterization of different acquisition models, and for which one can compare different models. In that sense, the operations defined above do not appear to warrant consideration in the basic framework.

We have assumed a set of requirements without dealing with them in any further detail. One area of possible consideration deals with requirements *semantics*. For example, one could require that the semantics of a requirement be unique with respect to other requirements. The formalism applied here permits such a specification, and while perhaps of general interest, we chose not to include it.[14]

14. The language *Nestor* includes some powerful operations for dealing with semantics of predicates and associated state data.

5.3 Participants

Acquisition is performed in a context that has multiple participants, each possibly assuming different roles. For example, one could consider

- end-user
- developer
- project manager

Another choice might be to consider

- end-user
- project manager
- developer agent
- prime contractor
- sub-contractor

In the latter case, the roles of prime contractor and sub-contractor are similar to that of the developer, which appears in the first set of alternatives. Furthermore, it is also possible to refine the roles of the project manager. For example, one could have a test manager, budget analyst, and so on.

Recognizing the many possible choices to delineate various roles on a project, we have chosen to remain silent about these roles in the specification of the framework. Of course, a particular set of roles could be included in the development of a particular acquisition model.

Another reason for not including the roles is based on a separation principle of *what* versus *who*. Our prime concern is what activities are performed, rather than who performs that activity. While we recognize the possibility of including a particular role, we believe that is outside the scope of the core framework.

The following provides some guidance, if one were interested in specifying participants in a specification of an acquisition model. We would begin by declaring a free type to represent a participant, namely:

[PARTICIPANT]

Next, we need to identify who the particular participants are. For example, from the perspective of a project office that is responsible for management of the acquisition, we could include the following:

Project_Manager, System_Engineering, Software_Engineering, System_Test: PARTICPANT

Project_Management_Staff$_{[t]}$: {Project_Manager, System_Engineering, Software_Engineering, System_Test}

From the developer perspective, we might wish to define the following set of participants:

Developer_Manager, Developer_System_Engineering: PARTICIPANT

Given the basic set of participants, there are now operations that we can perform on these basic declarations. For example, suppose we wanted to specify a joint Integrated Project Team, whose members were acquisition project managers and the manager of the developer effort. We could do this in the following manner:

Joint_Management_Team$_{[t]}$: {Project_Manager, Developer_Manager}

Also, for purposes of illustration, we can create a joint team for system engineering, namely:

Joint_System_Engineering_Team$_{[t]}$: {System_Engineering, Developer_System_Engineering}

Hence, given the basic type declaration for a participant, we can declare and combine them in ways that are appropriate to the particular concerns of the model. For example, suppose we were interested in including an operation in the acquisition model that specified the approval of some acquisition activity. This can be represented as a relation between participants and activities, as shown in Figure 5-1.

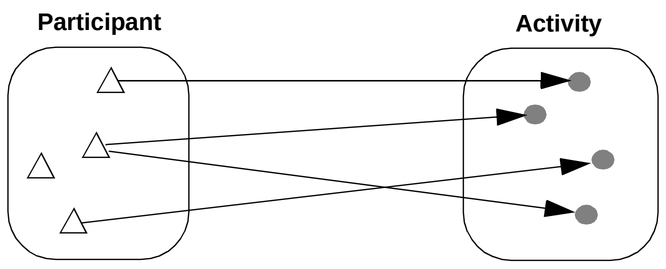


Figure 5-1: Development of Approval Relation

There are different ways in which the approval process can be modeled. For example, suppose we have the following model:

- Every activity must have at least one participant who can approve the activity (the relation is total over the range).

- A given participant may approve more than one activity (the relation is many-to).
- A given activity may be approved by only one participant (the relation is to-one).
- Not all participants need to have approval authority for an activity (the relation is partial over the domain).

The specification of a relation satisfying the characteristics above is as follows:

$$\begin{array}{l} \textbf{Approval\_Authority}: \text{Participants}_{[t]} \text{ <--|----- } \text{Activities}_{[t]} \\ \quad \exists\, p_i \in \text{Participants}_{[t]},\, a_j \in \text{Activities}_{[t]} \bullet \\ \quad\quad \textbf{Approval\_Authority}\ (p_i, a_j) \end{array} \qquad (5.1)$$

Given the specification of the above relation, we can declare those ordered pairs that are members of the relation. For example, suppose we were concerned with activities of requirements management and system development. The relevant relations could be declared as:

Approval_Authority (Project_Manager, Requirements_Management)
Approval_Authority (System_Engineering, System_Development)
Approval_Authority (Developer_System_Engineering, System_Development) (5.2)

Note that the above permits two different participants to have approval authority for the system development activity, namely either a project or developer agent who is responsible for system engineering. It does not, however, permit the overall project manager to have approval authority for system development.[15]

We may use the above notion of approval authority to refine the specification of what it means to initiate some acquisition activity. In the development of the framework we included an operation that specified the initiation of some acquisition activity (see Eq. (2.29) on page 23). We include the concept of approval authority as a pre-condition to initiating some activity; in other words, an activity cannot begin until there is a participant who has approval authority for the activity. Hence, we modify Eq. (2.29) as follows:

$$\begin{array}{l} \textbf{Initiate\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY; } t) \rightarrow \text{Boolean} \\ \quad |\!\rightarrow \\ \quad\quad \exists\, p_i \in \text{Participants}_{[t]} \bullet \\ \quad\quad\quad \textbf{Approval\_Authority}\ (p_i, a_j) \end{array} \qquad (5.3)$$

15. The degree to which this choice is appropriate will not be further speculated upon here!

Hence, Eq. (5.3) means that an acquisition activity cannot be initiated unless there is a participant who has approval authority for that particular activity. This is a reasonable approach to dealing with approval requirements for acquisition activities, although we recognize that other approaches are possible. For example, the project manager could have a “veto” power and ability to approve any acquisition process. If this is the desired approach, however, the specification must account for the application of a veto power by the acquisition manager.

5.4 Artifacts

As part of an acquisition there are many artifacts that are developed or used. We use the term artifact to denote some document or product that is developed or used in an acquisition. Examples of artifacts can include

- software development plan
- contract
- software requirements document
- standard
- COTS product[16]
- budget

Clearly, as indicated above, artifacts can be of different types and serve different purposes. Furthermore, artifacts can be used in different ways in the same acquisition.

We do not include artifacts in the specification of the framework. Our reason is that a specification of artifacts, relations among them, and predicates associated with their use is not fundamental in order to compare different acquisition models that can be created as an instance of the framework. That is, many different models can be used that employ artifacts, but the manner in which the artifacts are used, we believe, is not fundamental to the ability to characterize key differences

16. It may be more likely that COTS products, as well as standards, are treated in an independent manner; i.e., not as a subset of general artifacts. The choice to be made relates to the importance one wishes to attribute to standards and COTS products in an acquisition model.

among different acquisition models.[17]

It would be possible to include some general operations on artifacts in the framework. Toward this end, we could specify a data type

[ARTIFACT]

We could then specify a set of artifacts and perform operations on a general artifact like *create_artifact*, *modify_artifact*, and so on. A similar approach was taken in the way we dealt with requirements, for example (see Section 2.1.3 on page 7).

However, consider the case where one is interested in a specification of an acquisition model that places emphasis on the use of COTS products. While a COTS product can be viewed as a type of artifact, assume that we wish to treat it as a first-class object. In this case, we make the declaration:

[COTS_PRODUCT]

We can also declare a set whose elements are all COTS products (as opposed to also including artifacts of other types) as

$\text{COTS\_Products}_{[t]}$: {COTS_PRODUCT}

Taking the above approach allows us to concentrate on aspects of the acquisition model that are specific to the use of COTS products. For example, some predicates that we may find useful in describing an acquisition model that emphasizes COTS products include[18]

New_COTS_Product_Available (c_i: COTS_PRODUCT; t) $\rightarrow$ Boolean

COTS_Product_Evaluated (c_i: COTS_PRODUCT; t) $\rightarrow$ Boolean

COTS_Product_Acceptable (c_i: COTS_PRODUCT; t) $\rightarrow$ Boolean

17. One could make the same argument for requirements. However, we note the following important point: The manner in which a set of requirements is mapped onto different builds is indeed a key characteristic that allows comparison among different acquisition models. For example, a key difference between a waterfall model and an incremental model is the manner in which the requirements are mapped onto different instances of a system. Hence, requirements must be included in the framework in order to make the comparison among different acquisition models.

18. Further consideration of acquisition models that include COTS products can be found in [5].

COTS_Product_Upgraded (c_i: COTS_PRODUCT; t) $\longrightarrow$ Boolean

The predicates defined above represent only a small sample of what could be specified in a particular acquisition model. The important point is that the framework provides the necessary constructs to develop an approach for dealing with artifacts appropriate to a given acquisition model.

5.5 Entrance and Exit Criteria

There are cases where an acquisition includes a specified set of entrance and exit criteria. These criteria represent conditions that must be satisfied in order to either initiate or complete a specified activity, respectively. The following are examples of such criteria:

- There must be no high-priority defects in any software component prior to entering system testing.
- There must be no outstanding change requests against a requirements document in order to complete a requirements specification activity.

Specification and operations on entrance and exit criteria have not been included as part of the framework. This choice was made for two reasons:

1. In many cases the entrance and exit criteria are rather trivial. For example, it is often the case that the entrance criteria for an activity are associated with the completion of the previous activity. Similarly, the exit criteria for an activity may be principally associated with the completion of the activity. To include material at a lower level of specification was deemed unsuitable in the framework.
2. The framework already contains implicit mechanisms to incorporate entrance and exit criteria. For example, the predicate to delete an acquisition activity was defined by Eq. (2.12) on page 17:

   Delete_Activity (a_i: ACTIVITY; t') $\longrightarrow$ Boolean
   $|\!\Rightarrow$
   $$\underset{t' > t}{\text{Acq\_Activities}_{[t']} = \text{Acq\_Activities}_{[t]} \setminus \{a_i\}}$$

   Note that there is no pre-condition associated with the operation that deletes an acquisition activity. If one wanted to account for entrance criteria in the above predicate, the natural approach would be to add a pre-condition that specifies that the desired criteria must be satisfied.[19] Thus, the framework already provides mechanisms by which entrance

19. For example, one might reasonably argue that an acquisition activity should not be deleted if the activity is in progress.

and exit criteria may be included as part of a refinement of the framework in the context of a particular model.

We recognize, however, that there are indeed cases where treatment of entrance and exit criteria is important. Some examples of such cases were illustrated above. To specify such criteria in the framework, we introduce a criterion and then define relevant sets as

[CRITERION]
Entrance_Criteria$_{[t]}$: {CRITERION}
Exit_Criteria$_{[t]}$: {CRITERION}

We could have required that the sets *Entrance_Criteria* and *Exit_Criteria* be disjoint. However, we believe that in the context of a *framework*, as opposed to an instance of that framework, such a requirement would be overly constraining. In other words, there may be some criteria that are used both as entrance criteria and exit criteria.

It is appropriate to specify the relation between the criteria and acquisition activities. We assume the following general conditions apply to both entrance criteria and exit criteria:

- Each criterion must be related to at least one activity.
- A given criterion may apply to more than one activity.
- Not all activities need have criteria associated with them.
- A given activity may have multiple criteria.

A diagram of the relation between criteria and activities that satisfies the above conditions is shown in Figure 5-2.

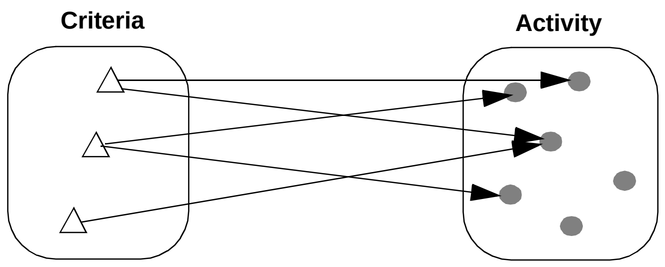


Figure 5-2: Relation Between Entrance and Exit Criteria and Activities

The relations between the criteria and acquisition activities, representing the above conditions, are:

Entrance_Criterion_Activity_Relation: Entrance_Criteria$_{[t]}$ <---|-> Acq_Activities$_{[t]}$
$\exists$ $e_i \in$ Entrance_Criteria$_{[t]}$, $a_j \in$ Acq_Activities$_{[t]}$
• **Entrance_Criterion_Activity_Relation** (e_i, a_j) (5.4)

Exit_Criterion_Activity_Relation: Exit_Criteria$_{[t]}$ <---|-> Acq_Activities$_{[t]}$
$\exists$ $e_i \in$ Exit_Criteria$_{[t]}$, a_j: Acq_Activities$_{[t]}$
• **Exit_Criterion_Activity_Relation** (e_i, a_j) (5.5)

In the above we have defined a relation for each of the relevant types of criteria.[20]

It is also helpful to introduce some operations on entrance and exit criteria. We would like to be able to specify that some criterion is satisfied at a time t. We write these in the following manner:

Entrance_Criterion_Satisfied (e_i: ENTRANCE_CRITERIA, a_j: ACTIVITY; t) $\rightarrow$ Boolean (5.6)

Exit_Criterion_Satisfied (e_i: EXIT_CRITERIA, a_j: ACTIVITY; t) $\rightarrow$ Boolean (5.7)

To determine whether, for example, some entrance criterion is *not* satisfied, we can simply negate the predicate in Eq. (5.6).

How can we incorporate the entrance and exit criteria in terms of the basic framework specification developed above? This is a fairly straightforward matter if we recognize that these criteria can be considered to be pre-conditions on other predicates.

For example, to initiate some acquisition activity, we need to include a pre-condition to the predicate **Initiate_Acquisition_Activity** defined in Eq. (2.29) on page 23. The statement of the pre-condition is that, in order to initiate an acquisition activity, all of the entrance criteria for that activity must be satisfied. We can represent this requirement as follows:

Initiate_Acquisition_Activity (a_j: ACTIVITY; t) $\rightarrow$ Boolean
$\mapsto$
$\forall$ $e_i \in$ Entrance_Criteria • **Entrance_Criterion_Activity_Relation** (e_i, a_j)
Entrance_Criterion_Satisfied (e_i: ENTRANCE_CRITERIA, a_j: ACTIVITY; t')
t' < t (5.8)

In a similar manner, we can develop the specification for the exit criteria in terms of the termination of an acquisition activity as follows:

20. It would be possible to write the relations in *Nestor* in terms of a generic and then instantiate the generic mapping for each case, but we have chosen not to follow that approach.

$$
\begin{array}{l}
\textbf{Terminate\_Acquisition\_Activity}\ (a_j\text{: ACTIVITY; t}) \rightarrow \text{Boolean} \\
\quad \mid\!\rightarrow \\
\qquad \forall\ e_i \in \text{Exit\_Criteria} \bullet \textbf{Exit\_Criterion\_Activity\_Relation}\ (e_{i\,,}\ a_j) \\
\qquad\quad \textbf{Exit\_Criterion\_Satisfied}\ (e_i\text{: EXIT\_CRITERIA, } a_j\text{: ACTIVITY; t'}) \\
\qquad\qquad t' < t
\end{array}
\qquad (5.9)
$$

Note that if there are no criteria associated with an activity, the above expressions are equivalent to those developed earlier. That is, for example, if there are no entrance criteria associated with an activity, then Eq. (5.4) is identical to Eq. (2.29).

As a means of summary, in Figure 5-3 on the next page we present a specification for treating entrance and exit criteria as part of the acquisition framework.

We emphasize that this represents a candidate specification for entrance and exit criteria. We believe that it is sufficiently general that it can be the basis for other possible refinements Some items of possible interest include the following:

- If some entrance criteria are satisfied when an acquisition activity is initiated, is it necessarily true that the same entrance criteria are also satisfied when the activity terminates?
- Can exit criteria for some activity be changed during the execution of that activity? In other words, should it be possible to *modify* the exit criteria?
- Can an exit criterion be deleted during the execution of its associated activity? If so, what should happen?
- During the execution of some activity, can new entrance criteria be added that are related to the activity in progress?

The above questions illustrate issues that may be worth considering as possible extensions to the framework. The questions also apply if one seeks to include entrance and exit criteria in the development of an acquisition model.

5.6 Phases

In some acquisition approaches one encounters the concept of an *acquisition phase*. Loosely speaking, a phase is a collection of related acquisition activities. There are different approaches to specifying a phase.

Theory Entrance_Exit_Criteria

```
[CRITERION]
Entrance_Criteria$_{[t]}$: {CRITERION}
Exit_Criteria$_{[t]}$: {CRITERION}

**Entrance_Criterion_Activity_Relation:** Entrance_Criteria$_{[t]}$ ---> Acq_Activities$_{[t]}$
    $\exists$ $e_i \in$ Entrance_Criteria$_{[t]}$, $a_j \in$ Acq_Activities$_{[t]}$
        $\bullet$ **Entrance_Criteria_Activity_Relation** ($e_i$, $a_j$)

**Exit_Criterion_Activity_Relation:** Exit_Criteria$_{[t]}$ ---> Acq_Activities$_{[t]}$
    $\exists$ $e_i \in$ Exit_Criteria$_{[t]}$, $a_j \in$ Acq_Activities$_{[t]}$
        $\bullet$ **Exit_Criterion_Activity_Relation** ($e_i$, $a_j$)

**Entrance_Criterion_Satisfied** ($e_i$: ENTRANCE_CRITERIA, $a_j$: ACTIVITY)
    $\rightarrow$ Boolean

**Exit_Criterion_Satisfied** ($e_i$: EXIT_CRITERIA, $a_j$: ACTIVITY) $\rightarrow$ Boolean

**Initiate_Acquisition_Activity** ($a_j$: ACTIVITY; t) $\rightarrow$ Boolean
    $\mapsto$
        $\forall$ $e_i \in$ Entrance_Criteria$_{[t]}$ $\bullet$ **Entrance_Criterion_Activity_Mapping** ($e_i$, $a_j$)
        **Entrance_Criterion_Satisfied** ($e_i$: ENTRANCE_CRITERIA, $a_j$: ACTIVITY)

**Terminate_Acquisition_Activity** ($a_j$: ACTIVITY; t) $\rightarrow$ Boolean
    $\mapsto$
        $\forall$ $e_i \in$ Exit_Criteria$_{[t]}$ $\bullet$ **Exit_Criterion_Activity_Mapping** ($e_i$, $a_j$)
            **Exit_Criterion_Satisfied** ($e_i$: EXIT_CRITERIA, $a_j$: ACTIVITY)
```

end Entrance_Exit_Criteria

Figure 5-3: Theory Component for Entrance and Exit Criteria

5.6.1 Sequential Phases

One approach to specifying an acquisition phase is to define it as a *sequence* of activities where the activities that comprise the phase are performed serially. To formalize this approach to a phase, we begin by defining a phase as

Phase$_{[t]}$: sequence {ACTIVITY}
#Phase > 0

We require that the number of activities in the phase must be greater than zero. Note that the above definition of a phase does not require that each activity be performed only once.

We next define the start and end of a phase as follows:

t_{start}, t_{stop}: Time
$t_{start} \rightarrow$ **Time_of** [**Initiate_Activity** (Head(Phase$_{[t]}$))]
$t_{stop} \rightarrow$ **Time_of** [**Terminate_Activity** (Tail(Phase$_{[t]}$))]

The start and stop times are obtained from the times when the first and last activities in the phase are initiated and terminated, respectively.[21] These are obtained by applying the function **Time_of** to a predicate.[22]

The requirement that only *one* activity may be performed at a time is specified by

$$\begin{array}{l} \forall\ t \bullet t_{start} \leq t \leq t_{stop} \\ \quad \exists_1\ a_i \in \text{Phase}_{[t]} \bullet \\ \qquad \textbf{Activity\_in\_Progress}\ (a_i;\ t) \end{array} \tag{5.10}$$

The above expression says that for all times between the start and stop time of a phase, there exists only *one* activity (denoted by the symbol $\exists_1$) that is in progress at any time.

It is also useful to define the property that an acquisition phase is well defined. By this we mean that the end of one activity corresponds to the beginning of the next activity. This is specified by the following:

$$\begin{array}{l} \forall\ i \in 1 \ldots \#\text{Phase}_{[t]} - 1 \\ \quad \textbf{Terminate\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY}) = \\ \qquad \textbf{Initiate\_Acquisition\_Activity}\ (a_{i+1}\text{: ACTIVITY})) \end{array} \tag{5.11}$$

In other words, for every acquisition activity in the sequence from the first to the next to last (as indicated by the use of the ellipsis) the termination of one activity coincides with the start of the next activity. This implies that there is no “idle time” between the end of one activity and the initiation of another activity.

21. When an acquisition phase is represented as a sequence, the operations *Head* and *Tail* return the first and last element in the sequence, respectively.

22. The function **Time_of** is a built-in function in the language *Nestor*. It is technically referred to as a *temporal projection operator*.

The preceding approach to an acquisition phase is illustrated graphically in Figure 5-4.

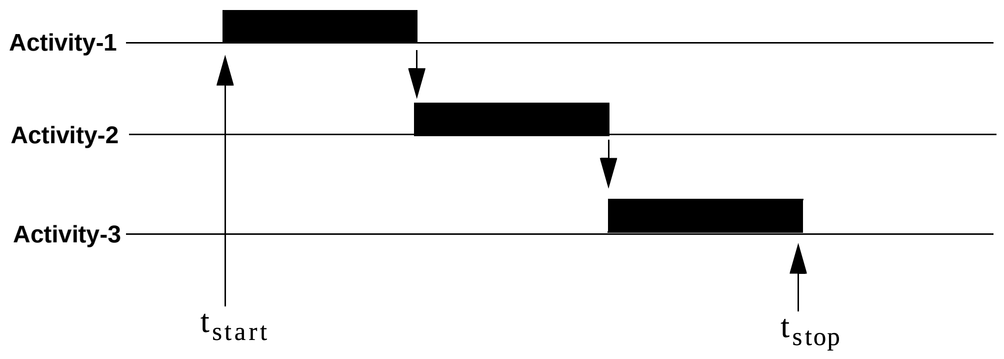


Figure 5-4: Illustration of an Acquisition Phase

Figure 5-4 shows three acquisition activities that are performed serially. The arrows between activities denote relations between the indicated activities. The starting and stopping times of the phase are also indicated. Figure 5-4 illustrates some of the criteria we specified for a serial phase; for example, the end of one activity corresponds to the beginning of the next activity. Note however, that the preceding development does not specify the number of times a given activity can be performed in a phase. That is, a particular activity could be performed one time or repeated times during an acquisition phase.

A summary of the formal specification of a sequential phase is shown in Figure 5-5 on the next page.

5.6.2 Other Choices

The sequential model of a phase is perhaps the most intuitive, but certainly other approaches are possible. One would expect a general requirement that there be transitive closure of the activities that comprise a phase, regardless of how the activities in the phase are executed on some interval.

One other choice for a specification of a phase is that it consist of a *set* of related activities, as opposed to a sequence of activities. This is tantamount to “a model within a model.” There is considerable freedom in how such an acquisition phase could be specified. Because it can be viewed as a general model, limited only by its start and stop of execution time, we shall not discuss this choice any further. Recognition of the “model in a model” context can be used to construct the desired features for what is included in a particular phase. It should be clear that a major characteristic of the use of phases in an acquisition deals with the timing of when activities are performed. Some further discussion concerning timing properties appears in Section 5.7.

```
Theory Serial_Phase

    [ACTIVITY]
    Phase$_{[t]}$: sequence {ACTIVITY}
    #Phase > 0

    $t_{start}$, $t_{stop}$: Time
    $t_{start} \rightarrow$ Time_of [Initiate_Activity (Head(Phase$_{[t]}$))]
    $t_{stop} \rightarrow$ Time_of [Terminate_Activity (Tail(Phase$_{[t]}$))]

    $\forall$ t $\bullet$ $t_{start} \leq t \leq t_{stop}$
        $\exists_1$ $a_i \in$ Phase$_{[t]}$
            Activity_in_Progress ($a_i$; t)

    $\forall$ i $\in$ 1... #Phase$_{[t]}$ - 1
        Terminate_Acquisition_Activity ($a_i$: ACTIVITY)) =
            Initiate_Acquisition_Activity ($a_{i+1}$: ACTIVITY))

end Serial_Phase
```

Figure 5-5: Specification of a Serial Phase

In closing, let us note that it is also possible to specify a relation between entrance and exit criteria (discussed in Section 5.5) and various formulations of an acquisition phase. For example, the material developed in Section 5.5 can be integrated with the specification of phases so that one achieves a specification of entrance and exit criteria for an acquisition phase.

5.7 Execution of Acquisition Activities

One of the differentiators among a set of models is their operational semantics. In particular, timing of the performance of various acquisition activities is a significant factor. In this section we expand on this topic. Our purpose is to illustrate how a formal approach can help in the specification of this aspect of an acquisition model.

5.7.1 Serial Models

First, suppose we wanted to define a *sequential* acquisition model. We can state this in the following manner:

Sequential Acquisition Model: An acquisition model is sequential on some interval [t, t'] iff only one acquisition activity may execute at any one time.

A formal definition of a sequential model can be obtained by considering the following predicate:

$$
\begin{array}{l}
\textbf{Sequential\_Acquisition\_Model}\ (t,\ t') \rightarrow \text{Boolean} \\
\quad \mapsto \\
\qquad \forall\ t^* \in [t,\ t'] \\
\qquad\quad \exists_1\ a_i \in \text{Acq\_Activities}_{[t]} \bullet \\
\qquad\qquad \textbf{Execute\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY; } t,\ t^*)
\end{array}
$$

This is stated as a conditional predicate, in that if there exists only one activity that may execute at a given time, the model is sequential.

A possibly useful inquiry function when developing a model is the ability to determine if two different acquisition activities are performed in a serial manner. This would mean that the termination of one activity would coincide with the beginning of another activity. Such a function can be written as:

$$
\begin{array}{l}
\textbf{Serial\_Activities}\ (a_i,\ a_j\text{: ACTIVITY}) \rightarrow \text{Boolean} \\
\quad \mapsto \\
\qquad \textbf{Instant}\ [\textbf{Terminate\_Acquisition\_Activity}\ (a_i\text{: ACTIVITY; } t)] = \\
\qquad\quad \textbf{Instant}\ [\textbf{Initiate\_Acquisition\_Activity}\ (a_j\text{: ACTIVITY; } t')]
\end{array}
$$

An example of the application of this acquisition activity serialization is to the specification of the traditional waterfall model [6], where some set of activities is performed serially. In fact, a formal specification of the waterfall model would have a serialization requirement placed on the execution of acquisition activities.

5.7.2 Parallel Models

The more interesting case is where parallelism is present in the execution of acquisition activities. To begin, we define two operations that will prove useful, namely

$$
\begin{array}{l}
\textbf{Acquisition\_Activity\_Start}\ (a_j\text{: ACTIVITY; } t) \rightarrow \text{Time} \\
\quad \Rightarrow \textbf{Instant}\ [\textbf{Initiate\_Acquisition\_Activity}\ (a_j\text{: ACTIVITY; } t)]
\end{array}
$$

The operation **Acquisition_Activity_Start** is of type *time*; the function **instant** is a projection operator that extracts the value of time for a predicate that has a single argument of type Time. Hence, the result expressed above simply provides the time some activity started, namely t.

In a similar manner we define the time at which an activity stops as

Acquisition_Activity_Stop (a_i: ACTIVITY; t) $\rightarrow$ Time
  $\Rightarrow$ **Instant** [**Terminate_Acquisition_Activity** (a_i: ACTIVITY; t)

To develop the specification for parallel execution in an acquisition model, first let us define a predicate that indicates that two activities execute in parallel. We do this as

```
Parallel_Activities ($a_i$, $a_j$: ACTIVITY) $\rightarrow$ Boolean
    $|\rightarrow$
        $\forall$ t* $\in$ [t, t'] $\bullet$ Execute_Acquisition_Activity ($a_i$: ACTIVITY; t, t')
            $\exists$ $a_j \in$ Acq_Activities $\bullet$
                Initiate_Acquisition_Activity ($a_j$: ACTIVITY; t*) $\vee$
                Terminate_Acquisition_Activity ($a_j$: ACTIVITY; t*)                    (5.12)
```

In other words, activities a_i and a_j are parallel if, during the execution of activity a_i, activity a_j either starts or stops.

It is possible to distinguish between different types of execution. The first we shall denote as *closed parallel execution*, and the second will be denoted as *open parallel execution*. These are illustrated in Figure 5-6.

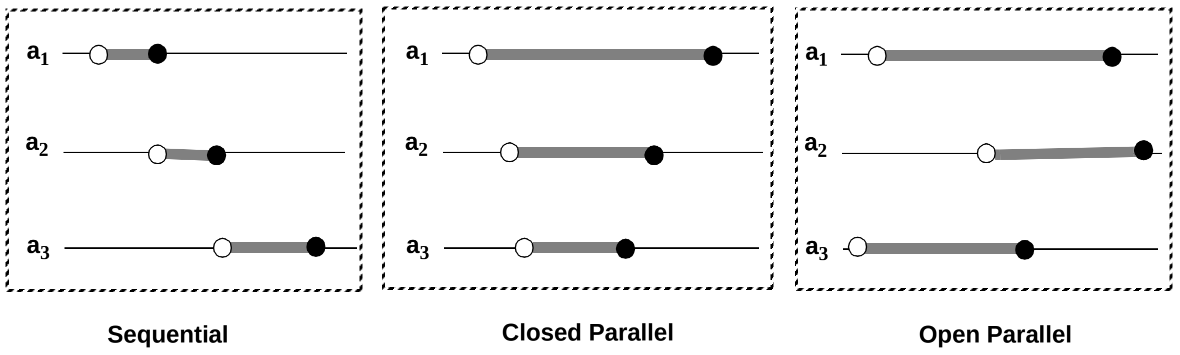


Figure 5-6: Types of Activity Execution

Each panel in Figure 5-6 shows the execution of three hypothetical acquisition activities. Time increases to the right along each line. The left-hand panel of Figure 5-6 shows the case of sequential execution.

5.7.2.1 Closed Parallel Models

The center panel in Figure 5-6 illustrates the case of closed parallel execution.We define this as follows:

> **Closed Parallel Execution:** An acquisition activity a_i executes in a closed parallel manner with respect to acquisition activity a_j iff the execution interval for activity a_j is wholly contained within the execution interval of activity a_i.

We have introduced the concept of an *execution interval* in the above definition. This is the set of times when an acquisition activity is executing.

We may develop a predicate to indicate whether two different acquisition activities are executed in a closed parallel manner:

```
Closed_Parallel_Activities ($a_i$, $a_j$: ACTIVITY) $\rightarrow$ Boolean
    $\mapsto$
        Acquisition_Activity_Start ($a_i$: ACTIVITY; t) $\leq$
            Acquisition_Activity_Start ($a_j$: ACTIVITY; t)
    $\wedge$
        Acquisition_Activity_Stop ($a_i$: ACTIVITY; t) $\geq$
            Acquisition_Activity_Stop ($a_j$: ACTIVITY; t)
                $a_i \neq a_j$                                        (5.13)
```

5.7.2.2 Open Parallel Execution

The other case of parallel execution of two acquisition activities is shown in the right panel of Figure 5-6. Note that in this case, some activities overlap in (the time of) their execution. We define this as

> **Open Parallel Execution:** An acquisition activity a_i executes in an open manner with respect to acquisition activity a_j iff (i) the activities execute in parallel, and (ii) the execution interval for activity a_j is not wholly contained within the execution interval of activity a_i.

We may formally define these concepts as follows. The specification for this case can be developed for the case of closed parallel execution. We have

```
Open_Parallel_Activities ($a_i$, $a_j$: ACTIVITY) $\rightarrow$ Boolean
    $\mapsto$
        Acquisition_Activity_Start ($a_i$: ACTIVITY; t) $\leq$
            Acquisition_Activity_Start ($a_j$: ACTIVITY; t)
    $\vee$
```

Acquisition_Activity_Stop (a_i: ACTIVITY; t) $\geq$
Acquisition_Activity_Stop (a_j: ACTIVITY; t)
$a_i \neq a_j$ (5.14)

The above result has been obtained by replacing an *and* operator by an *or* operator to join the pre-condition clause in Eq. (5.13).

The preceding discussion has focused on different ways parallel execution of different acquisition activities may be addressed in the development of an acquisition model.

5.7.3 Precedence Relations

There may be cases where it is desirable to specify precedence relations for the execution of acquisition activities. Examples of this include the following:

- An acquisition activity must be executed after some other activity.
- An acquisition activity must not be executed after some other activity.

It is possible to refine these to incorporate notions of strong and weak precedence. For example, strong precedence would be defined as the case where an acquisition activity must be executed immediately after some other activity. Weak precedence, on the other hand, is where there may be intervening activities between the (ordered) execution of two different activities.

It is possible to develop specifications for precedence relations based on results specified in the framework. For example, consider the case of weak precedence for some activities a_i and a_j where activity a_j must execute after activity a_i. A specification of this case would be as follows:

Weak_Activity_Precedence (a_i, a_j: ACTIVITY) $\rightarrow$ Boolean
$\mapsto$
$\forall$t $\bullet$ **Initiate_Acquisition_Activity** (a_j: ACTIVITY; t)
$\exists$ t' < t $\bullet$ **Terminate_Acquisition_Activity** (a_i: ACTIVITY; t')

Note that the preceding is a specification of weak precedence. It can easily be modified to account for the notion of strong precedence. It is also possible to specify the contrary position in regard to precedence. That is, one can specify that some acquisition activity must not precede some other activity.

We have not included operations in the framework for dealing with precedence of execution of acquisition activities. The simple reason for this is that it is possible to specify such conditions using operations that are already included in the framework specification. Hence, a model can easily construct the necessary operations from what is provided in the framework.[23]

5.8 Additional Considerations

The discussion in the preceding sections has largely focused on the functional extensions that may warrant consideration when developing an acquisition model based on the framework. There are other approaches that take a different approach than that of adding functionality. The following describes some approaches that are based on standard formal method approaches.

5.8.1 Promotion

Many of the predicates described in Section 2 are specified in the context of elements of a set. For example, in Section 2.3.3 we specified a predicate that indicated that a requirement was satisfied, that is

$$\textbf{Requirement\_Satisfied}\ (r_i\text{: REQUIREMENT}) \rightarrow \text{Boolean} \tag{5.15}$$

The above predicate is defined for one element of the set of requirements. We can *promote* the specification over the entire set of requirements by introducing a precondition in the following manner:

$$\begin{aligned}
&\textbf{All\_Requirements\_Satisfied} \rightarrow \text{Boolean}\\
&\quad \mapsto\\
&\qquad \forall\ r_i \in \text{Requirements}_{[t]} \bullet\\
&\qquad\quad \textbf{Requirement\_Satisfied}\ (r_i\text{: REQUIREMENT})
\end{aligned} \tag{5.16}$$

As indicated in Eq. (5.16), the predicate **All_Requirements_Satisfied** will be true iff each requirement in the set of requirements is satisfied.

Promotion in the preceding example in Eq. (5.16) is a simple and powerful technique that can be used to extend a basic specification over a larger scope. However, we have not included operations in the framework based on a promotion principle—such as Eq. (5.16)—for a simple reason: they can be directly inferred from the base specification. Hence, it is not required to include them in the framework, as they can be easily specified when needed.

23. We also do not account for the assertions that may be developed from the precedence considerations. For example, if activity a_i must precede activity a_j and activity a_j must precede activity a_k, then it follows that activity a_i must precede activity a_k.

5.8.2 Operational Refinement

The first form of refinement we will briefly describe is that of operation refinement. In this case, one adds additional details to some operation; the details are typically characteristic of the model one is developing.

Note that most of the operations defined in the framework do not have pre-conditions or post-conditions. This choice was made on purpose. The presence of either pre- or post-conditions can often represent additional levels of detail that we would prefer not be included in the framework because they are more specific to a particular model.

For example, consider the predicate to indicate that some requirement is satisfied. The meaning of such a predicate is self-explanatory (at least intuitively!), but it can be specified to a greater level of detail. There are different ways to state that a requirement has been satisfied, for example, through the use of

- analytic verification
- simulation
- system testing

For each of the different approaches, we can define a predicate.

$$\begin{aligned}
&\textbf{Requirement\_Analyzed}\ (r_i\text{: REQUIREMENT}) \rightarrow \text{Boolean}\\
&\textbf{Requirement\_Simulated}\ (r_i\text{: REQUIREMENT}) \rightarrow \text{Boolean}\\
&\textbf{Requirement\_Tested}\ (r_i\text{: REQUIREMENT}) \rightarrow \text{Boolean}
\end{aligned}$$

If, during the development of a particular acquisition model, one of the above approaches may be considered sufficient to represent that a requirement is satisfied, then we can easily write:

$$\begin{aligned}
&\textbf{Requirement\_Satisfied}\ (r_i\text{: REQUIREMENT}) \rightarrow \text{Boolean}\\
&\quad \mapsto\\
&\qquad \textbf{Requirement\_Analyzed}\ (r_i\text{: REQUIREMENT}) \vee\\
&\qquad\quad \textbf{Requirement\_Simulated}\ (r_i\text{: REQUIREMENT}) \vee\\
&\qquad\qquad \textbf{Requirement\_Tested}\ (r_i\text{: REQUIREMENT})
\end{aligned} \tag{5.17}$$

We will not argue the validity of the semantic choice that was made above, as that is a property of the model. The point is, however, it is possible to add additional details to the specification of a model that are derived from the base framework specification. Stated differently, refining the specifications of the operations that are included as part of the framework is one approach that can (and perhaps should) be followed in the development of a particular acquisition model.

5.8.3 Data Refinement

A second form of refinement is that known as data refinement. In this case, additional details about data are added to a specification. The result is that the specification becomes more concrete or, correspondingly, less abstract.

To illustrate how the principle of data refinement can be applied, consider the case where we introduced a (free) data type to represent an activity, namely

```
[ACTIVITY]
```

The fact that the data type is declared as above means that there is no specification of either the structure or semantics of the data type. Suppose, however, that one wanted to develop some details of the information associated with the data type for the activity. Some candidate elements that can be associated with an activity might include

- Last_Start_Time: The time when the activity was last started.
- Cumulative_Execution_Time: The total amount of time that the activity has taken.
- Current_Activity_State: An indication of the state, where the state could be null, in_progress, suspended, or completed. Such a model was used when we described the execution character of activities (see Section 2.4 on page 23).

The above data types can be specified as follows:

```
Last_Start_Time: Time
Cumulative_Execution_Time: Time

Activity_State: null | in_progress | suspended | completed
```

Recall that the type *time* is a built-in type in the language we are using. The specification of the activity state is represented as a choice, being one of the two values specified.

The above data types can now be used to refine the specification of an activity in the following manner:

$\text{ACTIVITY}_{[t]}$: {Last_Start_Time, Cumulative_Execution_Time, Activity_State}

The above declares that a particular activity is a *set* of the indicated elements. Using the above it is possible to perform operations such as

- Whenever an activity is initiated, the time that the activity was started will be saved.
- Whenever an activity is terminated, the cumulative execution time will be calculated and updated. Or, the most recent amount of execution time could be computed and saved.
- Whenever an activity is initiated, its state is set to the value of in_progress, and whenever the activity is suspended, its state is set to suspended.

Each of the preceding can be stated as part of the post-condition for the appropriate predicate. To do so we refine the notion of an ACTIVITY as follows:

ACTIVITY$_{[t]}$: sequence {Activity_State, Last_Start_Time,
Cumulative_Execution_Time}

The above simply takes the data type ACTIVITY and refines it so that an activity is now a sequence of data items. In particular, information about the state and the time the activity was last started will be maintained for each activity.

As a simple example of how the data refinement principle can be applied, consider the following modification to the predicate for initiation of an acquisition activity:

```
Initiate_Acquisition_Activity (a: ACTIVITY; t) → Boolean
    ⇒
        a.Activity_State = in_progress
        a.Last_Start_Time = t
        a.Cumulative_Execution_Time = 0
```

Note the use of the dotted notation in the above expression as a selector mechanism for the specified activity. The result is that the operation **Initiate_Acquisition_Activity** sets the value of the activity state to the value of in_progress, sets the time when activity was started to the current time, and initializes the cumulative execution time to zero.

The specification of the framework has not included any data refinement operations. To do so would be equivalent to defining data representation and semantics that would be imposed on all models derived from the framework. Clearly such a choice would be an over-specification. It is the choice of the model developer as to how such data refinement would be used in a particular model development.

6 Summary

In this report we have developed a general framework for the specification of acquisition models. The framework is intended to be minimalist in nature and is based on acquisition activities, events, requirements, and instances of a system as well as relations between these elements.

The value of an acquisition framework is that one can develop acquisition models that are based on the framework, possibly with extensions. This then allows one to gain a deeper understanding of the behavior of the model, as well as to compare different acquisition models.

We leave the reader with a final comment. The world moves ever more quickly toward the perspective of *systems of systems*. The interoperability of such systems is now a major issue in the ability to accomplish an overriding mission. Interoperability among systems is regarded as a technical issue, and rightly so. But there is another question that is very relevant, and we illustrate it in the following.

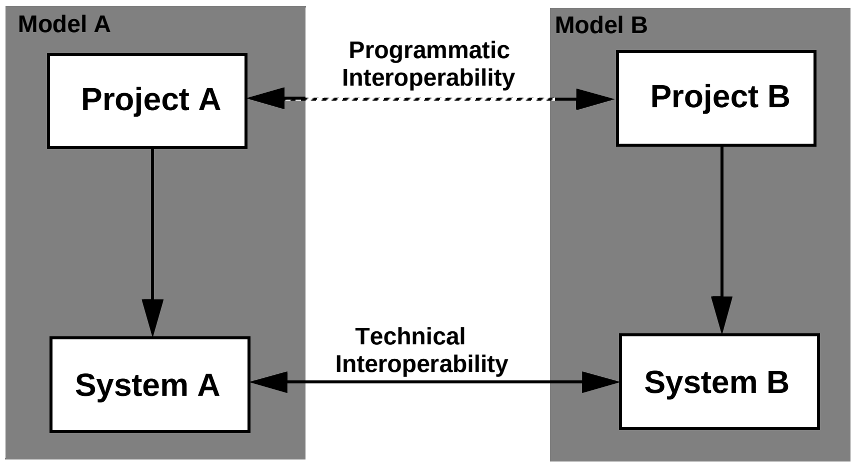


Figure 6-1: Multi-Project Acquisition

Figure 6-1 shows two acquisition projects, denoted A and B, whose acquisition is based on models A and B, respectively. It is required that the systems produced, denoted A and B, are required to interoperate, shown by the solid line at the bottom in the figure.[24]

[24] A diagram such as that presented in Figure 6-1 is known as a *four-corner diagram.*

Our final comment, then, is really a challenge: What does it mean, formally, to *integrate* individual acquisition models, in order to achieve *programmatic interoperability?* This is a question of serious concern, as well as ongoing work.

References

[1] Boehm, Barry W. “A Spiral Model of Software Development and Enhancement.” *IEEE Computer 21*, 5 (May 1988): 61-72.

[2] Department of Defense. DoD Instruction 5000.2, *Operation of the Defense Acquisition System*. Washington, D.C.: June 2001.

[3] Department of Defense. DoD Interim Regulation 5000.2-R, *Mandatory Procedures for Major Defense Acquisition Programs and Major Automated Information systems Acquisition Programs*. Washington, D.C.: June 2001.

[4] International Organization for Standardization, ISO/IEC 12207: *Information Technology—Software life-cycle processes,* Geneva, Switzerland, 1997).

[5] Meyers, B. Craig & Oberndorf, Patricia. *Managing Software Acquisition: Open Systems and COTS Products*. Reading, Mass.: Addison-Wesley, 2001.

[6] Royce, Winston, W. “Managing the Development of Large Software Systems,” 1-9. *Proceedings of IEEE WESCON*, August 1970. San Francisco: Institute of Electrical and Electronics Engineers, 1970.

Appendix A Summary of Framework Specification

In this Appendix we provide a summary of the mathematics used to describe the acquisition framework presented in Section 2. The base specification is provide below and it serves to reference other elements of the specification which appear on the following pages.

```
Theory Acquisition_Framework

    with  Framework_Activities,
          Framework_Internal_Events,
          Framework_External_Events,
          Framework_Requirements,
          Framework_System,
          Framework_Relations,
          Framework_Execution,
          Framework_Execution_Concurrency,
          Framework_Specification_Completeness,
          Framework_Execution_Completeness

end Acquisition_Framework
```

Theory Framework_Activities

```
    [ACTIVITY]
    $Acq\_Activities_{[t]}: \{ACTIVITY\}$
    $\#Acq\_Activities_{[t]} > 0$

    **Activity_Relation**: $Acq\_Activities_{[t]}$ <-----> $Acq\_Activities_{[t]}$
        $\forall a_i, a_j \in Acq\_Activities_{[t]}$
            • **Activity_Relation** $(a_i, a_j)$
                $a_i \neq a_j$

    **Add_Activity** $(a_i: ACTIVITY; t, t') \longrightarrow$ Boolean
        $|\Rightarrow$
            $Acq\_Activities_{[t]} = Acq\_Activities_{[t']} \cup \{a_i\}$
            $t > t'$

    **Modify_Activity** $(a_i: ACTIVITY; t) \longrightarrow$ Boolean

    **Delete_Activity** $(a_i: ACTIVITY; t, t') \longrightarrow$ Boolean
        $|\Rightarrow$
            $Acq\_Activities_{[t]} = Acq\_Activities_{[t']} \setminus \{a_i\}$
                $t > t'$

    **Activity_Started** $(a_i: ACTIVITY; t) \longrightarrow$ Boolean

    **Activity_in_Progress** $(a_i: ACTIVITY; t) \longrightarrow$ Boolean

    **Activity_Suspended** $(a_i: ACTIVITY; t) \longrightarrow$ Boolean

    **Activity_Completed** $(a_i: ACTIVITY; t) \longrightarrow$ Boolean
```

end Framework_Activities

Theory Framework_Internal_Events

```
    [INTERNAL_EVENT]
    Internal_Events$_{[t]}$: {INTERNAL_EVENT}

    Create_Internal_Event (e: INTERNAL_EVENT; t, t') $\rightarrow$ Boolean
        $\models$
            Internal_Events$_{[t]}$ = Internal_Events$_{[t']}$ $\cup$ {e}
                t > t'

    Modify_Internal_Event (e: INTERNAL_EVENT; t) $\rightarrow$ Boolean

    Delete_Internal_Event (e: INTERNAL_EVENT; t, t') $\rightarrow$ Boolean
        $\models$
            Internal_Events$_{[t]}$ = Internal_Events$_{[t']}$ \ {e}
                t > t'

    Internal_Event_Occurred (e: INTERNAL_EVENT; t) $\rightarrow$ Boolean

    Generate_Internal_Event (e: INTERNAL_EVENT; t) $\rightarrow$ Boolean
        $\models$ Internal_Event_Occurred (e: INTERNAL_EVENT; t)
```

end Framework_Internal_Events

Theory Framework_External_Events

[EXTERNAL_EVENT]
$External\_Events_{[t]}$: {EXTERNAL_EVENT}

External_Event_Occurred (e: EXTERNAL_EVENT; t) $\rightarrow$ Boolean

end Framework_External_Events

Theory Framework_Requirements

[REQUIREMENT]
$\text{Requirements}_{[t]}$: {REQUIREMENT}
$\#\text{Requirements}_{[t]} > 0$

Add_Requirement (r_i : REQUIREMENT; t, t') $\rightarrow$ Boolean
$\models$
$\text{Requirements}_{[t]} = \text{Requirements}_{[{}^{\prime}t]} \cup \{r_i\}$
$t > t'$

Modify_Requirement (r_i: REQUIREMENT; t) $\rightarrow$ Boolean

Delete_Requirement (r_i: REQUIREMENT; t, t') $\rightarrow$ Boolean
$\models$
$\text{Requirements}_{[t]} = \text{Requirements}_{[t']} \setminus \{r_i\}$
$t > t'$

Requirement_Satisfied (r_i: REQUIREMENT) $\rightarrow$ Boolean

Requirement_Dependency (r_i, r_j: REQUIREMENT; t) $\rightarrow$ Boolean

$\forall\, r_i \in \text{Requirements}_{[t]} \bullet$
Requirement_Dependency (r_i, r_j: REQUIREMENT; t)
$\Rightarrow$
$\neg$ **Requirement_Dependency** (r_j, r_i: REQUIREMENT; t)

end Framework_Requirements

```
Theory Framework_System

    [SYSTEM]
    System$_{[t]}$: {SYSTEM}

    System_Dependency ($s_i$, $s_j$: SYSTEM; t) $\rightarrow$ Boolean

    Requirement_Satisfied ($r_i$: REQUIREMENT, $s_j$: SYSTEM; t) $\rightarrow$ Boolean

    System_Instance_Satisfied ($s_i$: SYSTEM; t) $\rightarrow$ Boolean
        $\mapsto$
            $\forall\ r_j \in$ Requirements$_{[t]}$ $\bullet$
                Requirements_Mapping ($r_j$, $s_i$) $\bullet$
                    Requirement_Satisfied ($r_j$; t)

end Framework_System
```

Theory Framework_Relations

Internal_Event_Activity_Relation: Internal_Events$_{[t]}$ <---|->Acq_Activities$_{[t]}$

$\forall$ $e_i \in$ Internal_Events$_{[t]}$ • $\exists$ $a_j \in$ Acq_Activities$_{[t]}$

• **Internal_Event_Activity_Relation** ($e_{i\,,}\, a_j$)

External_Event_Activity_Relation: External_Events$_{[t]}$ ---> Acq_Activities$_{[t]}$

$\exists$ $e_i \in$ External_Events$_{[t]}$, $a_j \in$ Acq_Activities$_{[t]}$

• **External_Event_Activity_Relation** ($e_{i\,,}\, a_j$)

Requirements_Mapping: Requirements$_{[t]}$ <----> System$_{[t]}$

$\forall$ $r_i \in$ Requirements$_{[t]}$ • $\forall$ $s_j \in$ System$_{[t]}$

• **Requirements_Mapping** ($r_{i\,,}\, s_j$))

end Framework_Relations

Theory Framework_Execution

```
    **Initiate_Acquisition_Activity** ($a_i$: ACTIVITY; t) $\longrightarrow$ Boolean
        $|\rightarrow$
            $\neg$ **Activity_in_Progress** ($a_i$: ACTIVITY; t)
        $|\Rightarrow$
            **Activity_in_Progress** ($a_i$: ACTIVITY; t)

    **Suspend_Acquisition_Activity** ($a_i$: ACTIVITY; t) $\longrightarrow$ Boolean
        $|\rightarrow$
            $\exists$ t' < t $\bullet$ **Initiate_Acquisition_Activity** ($a_i$: ACTIVITY; t')
        $|\Rightarrow$
            $\neg$ **Activity_in_Progress** ($a_i$: ACTIVITY; t)

    **Resume_Acquisition_Activity** ($a_i$: ACTIVITY; t) $\longrightarrow$ Boolean
        $|\rightarrow$
            $\exists$ t' < t $\bullet$ **Suspend_Acquisition_Activity** ($a_i$: ACTIVITY; t')
        $|\Rightarrow$
            **Activity_in_Progress** ($a_i$: ACTIVITY; t)

    **Terminate_Acquisition_Activity** ($a_i$: ACTIVITY; t) $\longrightarrow$ Boolean
        $|\rightarrow$
            $\exists$ t' < t $\bullet$ **Initiate_Acquisition_Activity** ($a_i$: ACTIVITY; t')
        $|\Rightarrow$
            $\neg$ **Activity_in_Progress** ($a_i$: ACTIVITY; t)

    **Execute_Acquisition_Activity** ($a_i$: ACTIVITY; t, t') $\longrightarrow$ Boolean
        $|\rightarrow$
            **Initiate_Acquisition_Activity** ($a_i$: ACTIVITY; t)
            **Terminate_Acquisition_Activity** ($a_i$: ACTIVITY; t')
                    t' > t
```

end Framework_Execution

Theory Framework_Execution_Concurrency

Concurrency_Permitted (a_i, a_j: ACTIVITY) $\rightarrow$ Boolean

Concurrency_Prohibited (a_i, a_j: ACTIVITY) $\rightarrow$ Boolean

Concurrency_Required (a_i, a_j: ACTIVITY) $\rightarrow$ Boolean

end Framework_Execution_Concurrency

Theory Framework_Specification_Completeness

$\text{Internal\_Events}_{[t]} \cap \text{External\_Events}_{[t]} = \varnothing$

end Framework_Specification_Completeness

Theory Framework_Execution_Completeness

```
∀ $a_i \in$ Acq_Activities$_{[t]}$
    ∃ $t_1 \geq t_{start}$ • **Initiate_Acquisition_Activity** ($a_i$: ACTIVITY; $t_1$) ∧
        ∃ $t_2 \leq t_{stop}$ • **Terminate_Acquisition_Activity** ($a_i$: ACTIVITY; $t_2$)
            $t_{start} < t_{stop}$

∀ $e_i \in$ External_Events$_{[t]}$ •
    **Time_of [External_Event_Occurred** ($e_i$: EXTERNAL_EVENT; t)**]** $\in [t_{start}, t_{stop}]$
        ∃ $a_j$: ACTIVITY •
            **External_Event_Activity_Relation** ($e_{i\,,}\, a_j$)
        ∧
            **Initiate_Acquisition_Activity** ($a_j$: ACTIVITY; t)
                $t \in [t_{start}, t_{stop}]$

∀ $e_i \in$ Internal_Events$_{[t]}$ •
    **Time_of [Internal_Event_Occurred** ($e_i$: INTERNAL_EVENT; t)**]** $\in [t_{start}, t_{stop}]$
        ∃ $a_j$: ACTIVITY •
            **Internal_Event_Activity_Relation** ($e_{i\,,}\, a_j$)
        ∧
            **Initiate_Acquisition_Activity** ($a_j$: ACTIVITY; t')
                $t \in [t_{start}, t_{stop}]$

∀ $r_i \in$ Requirements$_{[tstop]}$ •
    **Requirement_Satisfied** ($r_i$; $t_{stop}$)
```

end Framework_Execution_Completeness

Appendix B A Quick Tour of the Language *Nestor*

The specification of the acquisition framework presented in this paper is based on a language called *Nestor*. This language is being developed to express the semantics of dynamic systems. We provide here a very brief description of a subset of *Nestor* that is used in this report. It is hoped that such a description will allow readers who are not familiar with these concepts to gain some understanding of the mathematical approach we have taken.

B.1 Overview

An overview of the language *Nestor* and a process perspective for how it is applied is presented in Figure B-1.

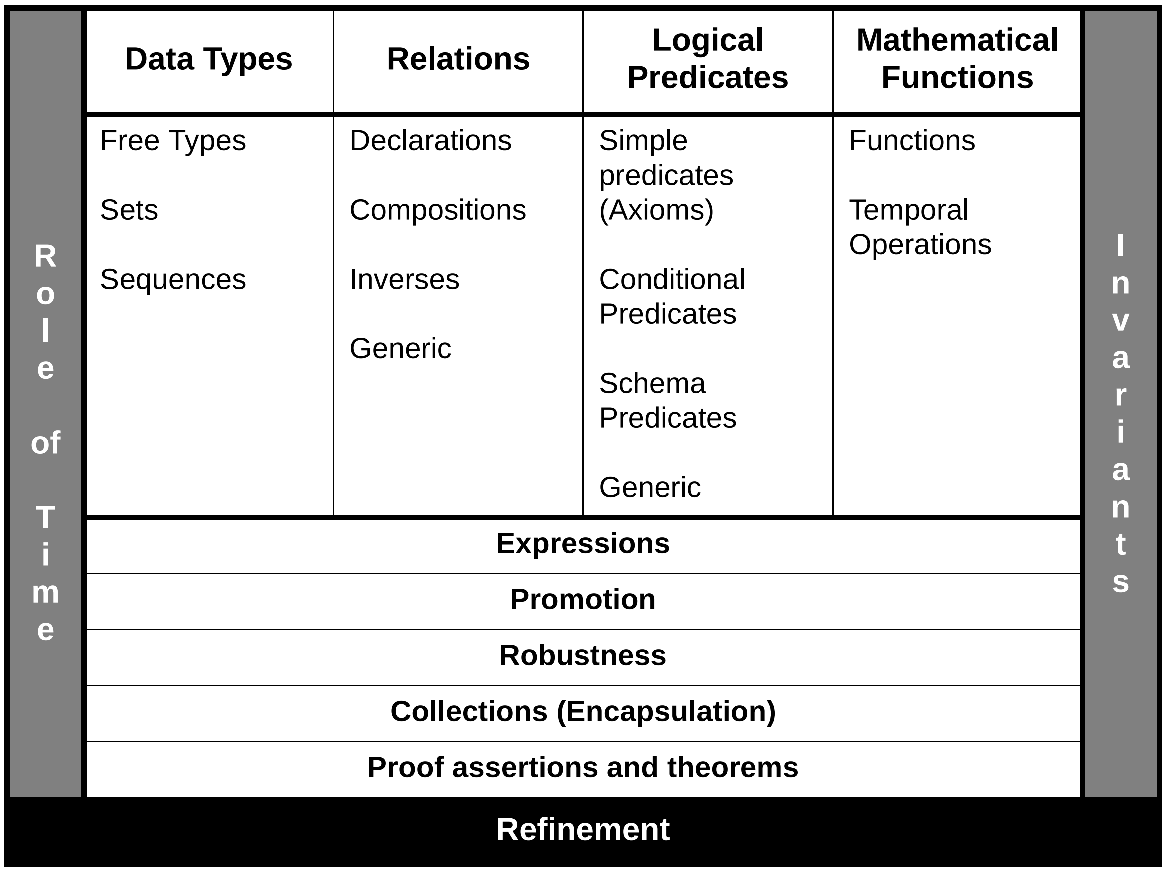


Figure B-1: Process for Specification Development

The four columns of the first row in Figure B-1 represent the main elements of a *Nestor* specification, namely, data types, relations, logical predicates, and mathematical functions. These can be viewed as building blocks to create a specification. In general the process of developing a specification involves the following activities:

- *Basic components:* The development of a specification typically begins with specifying basic components. These may include data types, relations, logical predicates, and mathematical functions.
- *Expressions:* The basic specification components can be used to develop expressions. For example, one could write a logical expression that involved relations between sets, as well as preconditions specified in terms of a mathematical function.
- *Promotion:* Promotion is the process in which some statement, over a primitive data type, is generalized to other members of the data type. For example, an expression may be developed for an element of the set and then promoted to all members of the set.
- *Robustness:* A complete specification must account for possible erroneous conditions, either in data or from an operational perspective. Robustness refers to the set of activities performed to handle such situations.
- *Collections:* A collection is a syntactic mechanism that allows one to encapsulate basic components and expressions. Collections may be used to partition different aspects of a specification.
- *Proof assertions:* It is possible to specify a theory construct that is another form of encapsulation mechanism. The components that make up the theory specification can include various statements. The theory construct enables one to make assertions about the theory. For example, given some theory and an arbitrary statement, are there inconsistencies present?

For each of the activities described above to develop a specification, there are two considerations that apply. These are shown by the shaded first and last columns of Figure B-1.

- *Role of time:* A *Nestor* specification has explicit built-in support for time, as well as temporal operations on predicates and functions with respect to time. We provide such capabilities because the envisioned domain of this language is for dynamic systems.
- *Invariants:* There are cases where there may be a property over data, or operations, that must be invariant. For example, if some set is required to have less than a specified number of members, this would be expressed as an invariant. Such invariants can be stated at different levels of a specification.

It must be emphasized that the development of a specification typically involves a *refinement* process. That is, from a given level of specification, one may develop further aspects of that specification by considering various refinement techniques. In *Nestor* these include data refinement, operation refinement, and temporal refinement. Each successive refinement adds additional detail to the specification.

In the sections following, we briefly describe some of the elements of the language that are particularly relevant to the development of the specification in this report.

B.2 Data Types

B.2.1 Free Types

A free type is a data type that is specified without any further consideration. Free types are frequently used in the construction of other data types.[25] Such types are often used to denote a basic type in the language and are enclosed in brackets. For example, to declare a free type to denote a requirement, we define it as:

```
[REQUIREMENT]
```

Multiple free types can be declared in one statement. Thus, to declare data types for entrance and exit criteria, we may write this as:

```
[ENTRANCE_CRITERIA, EXIT_CRITERIA]
```

B.2.2 Sets

A set is a collection of objects that have some relation to each other. In *Nestor* a set is represented by enclosing curly braces around the data element that comprises the set. For example, we introduced above the notion of a free type to represent a requirement. If we then wanted to be able to speak of a set of requirements, we can do this as follows:

```
Requirements: {REQUIREMENT}
```

Or, to denote sets of entrance and exit criteria, we can simply declare them as:

```
Entrance_Criteria: {ENTRANCE_CRITERIA}
Exit_Criteria: {EXIT_CRITERIA}
```

25. More specifically, a free type denotes a set of all possible values of the defined instance.

The cardinality of a set is the number of members in the set, and is denoted by the symbol “#”. Thus, #Requirements denotes the number of requirements.

If an element x is a member of a set X, then we write this as $x \in X$. Conversely, if x is not a member of the set X, then this is written as $x \notin X$.

The usual mathematical operations on sets are provided. These are summarized in Table B-1.

Name	Symbol	Example	Meaning
Null set	$\varnothing$	$A = \varnothing$	The set A has no members.
Union	$\cup$	$C = A \cup B$ $C = \{c \mid c \in A \vee c \in B\}$	The members of the set C are members of either the set A or the set B.
Intersection	$\cap$	$C = A \cap B$ $C = \{c \mid c \in A \wedge c \in B\}$	The members of the set C are members of both sets A and B.
Difference	$\backslash$	$C = A \backslash B$ $C = \{c \mid c \in A \wedge c \notin B\}$	The members of the set C are those elements that are a member of set A and not a member of set B.

Table B-1: Set Notation

B.2.3 Sequences

A sequence is a list of ordered elements. We might speak of a list of names or a list of addresses. In an acquisition context, we might have a sequence of activities, and this might be declared as

```
[ACTIVITY]
Activity_List: Seq {ACTIVITY}
```

Because a sequence is ordered, it allows us to reference the first or last element in the sequence. The first element is known as the *head* of the sequence and denoted head {Activity_List}. Similarly, the last item in the sequence, known as the tail, is denoted tail {Activity_List}. The number of elements in the sequence is denoted #{Activity_List}.

B.2.4 Time

Because *Nestor* is being developed to handle systems that are inherently dynamic in nature, it is important to deal with operations related to time. Hence, it is necessary to declare data types of type time. The word “time” is built-in for this purpose; thus, for example, the following declares identifiers t and t’ to be of type time:

t, t’: Time

B.2.5 Dynamic Data Typing

A further consideration regarding the desire to specify a dynamic system is to recognize that values of data types may change in time. The preceding causes us to account for what we call dynamic data typing: a declaration of a data object that may change in time. This is achieved by including subscripted brackets in the declaration of a data type. For example, if the set of requirements that we introduced earlier may change in time, this can be specified as

$\text{Requirements}_{[t]}$: {REQUIREMENT}

The presence of a subscripted [t] denotes a dynamic data type.[26] There are two reasons for the explicit provision of dynamic data types. First, it is a simple reminder to the specification developer (and the reader) that the values of the type may change with time. Second, it allows for operations to be performed using the dynamic data types at different instants of time.[27] This ability turns out to be a very powerful means of reasoning over the dynamic behavior of a specification.

For example, consider the following:

r: REQUIREMENT

$\text{Requirements}_{[t']} = \text{Requirements}_{[t]} \cup \{r\}$
$t' > t$

The preceding declares some requirement called r. We then define the set of requirements at a time t’ to be the union of requirements at an earlier time t and the requirement r. Note that we require t’ to be greater than t.

26. The presence of a parameter within the subscripted brackets when a data type is declared is optional and carries no semantic importance.

27. Hence, one may think of a dynamic data type as a family of instances of a basic data type that are mapped onto time. For example $\text{Requirement}_{[t]}$ and $\text{Requirement}_{[t']}$ denotes the set of requirements at two different instants of time t and t’. It is in this sense that a dynamic data type may be considered to be a mapping of a base type onto instants of time.

B.3 Relations

Relations are used to specify the association, or coupling, of two sets. For example, in the framework there is a set of internal events, i.e., those events that are raised within the scope of the acquisition project. There is also a set of acquisition activities. These are declared as follows:

```
[INTERNAL_EVENT]
Internal_events: {INTERNAL_EVENT}

[ACTIVITY]
Acquisition_Activities: {ACTIVITY}
```

It is desirable to specify the fact that there is a relation from an internal event to an acquisition activity. Frequently, this notion is depicted in a diagram, such as the following:

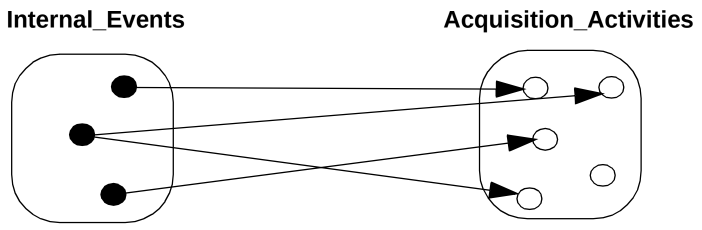


Figure B-2: Relating Internal Events to Acquisition Activities

The presence of the arrows in Figure B-2 denotes the relation between the internal events and the acquisition activities.

Formally, a (binary) relation is a set of ordered pairs. The relation is characterized by a set of source elements (the internal events in Figure B-2) known as the *domain*. It is also characterized by a set of destination elements (the acquisition activities in Figure B-2) known as the *range*.

There are two points about the number of elements in the domain and range of a relation that are relevant. The first point concerns whether the relation is total or partial, and is defined as follows:

- If the relation is defined for all elements of the domain, it is said to be *total* over the range. Conversely, if the relation is only defined for some elements of the domain, then the relation is said to be *partial* over the domain.
- A similar remark applies to the range: If the relation is defined for all elements of the range, it is total over the range. Conversely, if

the relation is defined for some of the elements in the range, it is partial over the range.

With reference to Figure B-2, the relation between internal events and acquisition activities is total over the domain (of internal events) and partial over the range (of acquisition activities).

The second point concerns the number of elements of the source and destination sets that are a member of the relation. These are distinguished by these properties:

- If there is at least one element of the domain that is related to more than one element in the range, then the relation is *many-to*.
- Conversely, if each element in the domain appears only once in the relation, the relation is *one-to*.

A similar remark applies to the range:

- If a member of the range can be related to more than one element in the domain, then the relation is *to-many*.
- Conversely, if a member of the range can appear only once in the relation, then the relation is *to-one*.

The relation indicated in Figure B-2 is many-to-one, total over the domain, and partial over the range. Although the use of figures to convey the notion of a particular relation is interesting, note that such figures do not substitute for the mathematical specification of the relation!

The preceding concepts can be combined to give different characteristics of a relation. For example, a relation may be many-to-one and total over both the domain and range.

When a relation is written in *Nestor*, it is necessary to account for the above concepts. For example, we will use the relation between internal events and acquisition activities to illustrate how relations are written. In this case, the relation would be declared as

$$\begin{aligned}&\textbf{Internal\_Event\_Activity\_Relation}\text{: Internal\_Events} \text{ <----|-> } \text{Acquisition\_Activities}\\ &\quad \forall\, e_i \in \text{Internal\_Events} \bullet \exists\, a_j \in \text{Acquisition\_Activities}\\ &\qquad \bullet\ \textbf{Internal\_Event\_Activity\_Relation}\ (e_{i,}\ a_j)\end{aligned} \qquad \text{(B.3)}$$

The following is an interpretation of the relation shown above:

- The first line declares the
  - name of the relation (**Internal_Event_Activity_Relation)**,
  - set that represents the domain (Internal_Events)

- set that represents the range (Acquisition_Activities)
- characterization of the relation as denoted by the symbol “<----|->”. The structure of this symbol is interpreted as follows:
  - The presence (absence) of the “<” on the left-hand side indicates that the relation is many-to (one-to) over the domain.
  - The presence (absence) of the symbol “>” on the right-hand side indicates that the relation is to-many (to-one).
  - the presence (absence) of a “|” on the left-hand side indicates that the relation is partial (total) over the domain.
  - the presence (absence) of the “|” on the right-hand side indicates that the relation is partial (total) over the range.

- The second line identifies the parameters in the relation, namely e_i and a_j. If a parameter is preceded by a universal quantifier symbol ($\forall$), it means the relation is total with respect to that parameter. If the relation is partial with respect to a parameter, then the parameter is preceded by an existential quantifier ($\exists$). The symbol “•” is read as “such that.”
- The third line repeats the name of the declaration with its associated parameters.[28]

Note that there is a certain amount of redundancy in the declaration of a relation; that is, whether the relation is partial or total is indicated by the symbol characterizing the relation (e.g., <----|->) and the operator that appears on the second line of the declaration. This choice is purposeful, as the symbol (<----|->) is a suggestive reminder of the character of the relation.

B.4 Predicates

Predicates are the means to specify operations in the language *Nestor*. A predicate can assume either the value true or false. There are four basic types of predicates. The first type of predicate is simply a declaration. For example, we can specify a predicate that indicates that some requirement is satisfied:

Requirement_Satisfied (r: REQUIREMENT) $\rightarrow$ Boolean

28. A verbal statement of the relation specified in Eq. (B.3) might be as follows: “There is a relation called *Internal_Event_Activity_Relation* that is from internal events to acquisition activities. The relation is many-to-many, total over the domain (of internal events) and partial over the range (of acquisition activities).”

In the above expression, we assume that the parameter r is of type requirement. The predicate can assume a Boolean value, that is, either true or false. As another example, we used a simple predicate to declare that some acquisition activity was in progress. Such a predicate was developed in Eq. (2.13) on page 18, namely

Activity_in_Progress (a_i: ACTIVITY; t) $\longrightarrow$ Boolean

Note the presence of the symbol "t" in the above predicate. When a predicate is declared, the parameters may include those of type time. Such parameters are identified by following the semi-colon in the predicate declaration. More than one parameter of type time may be present. Time is a built-in data type to the language *Nestor*.

The second type of predicate is a *conditional predicate* because it contains a clause that defines the condition(s) that must be satisfied in order for the predicate to assume the value True. For example, suppose we wanted to define an operation to initiate some acquisition activity at a time t. What is the pre-condition for such an operation? We suggest that an acquisition activity cannot be initiated unless it is not already in progress. This can be specified as follows:

Initiate_Acquisition_Activity (a_i: ACTIVITY; t) $\longrightarrow$ Boolean
$\quad|\!\longrightarrow$
$\qquad\neg$ **Activity_in_Progress** (a_i: ACTIVITY; t)

The symbol "$|\!\longrightarrow$" denotes the beginning of a pre-condition clause. The symbol "$\neg$" represents negation.

The third form of predicate is called an *unconditional predicate*. In this case there is *only* a post-condition clause present. We used a predicate of this type in Eq. (2.17) on page 19 where we declared a predicate to create an internal event:

Create_Internal_Event (e: INTERNAL_EVENT; t) $\longrightarrow$ Boolean
$\quad|\!\Rightarrow$
$\qquad \text{Internal\_Events}_{[t']} = \text{Internal\_Events}_{[t]} \cup \{e\}$
$\qquad\quad t' > t$

The parameter e is of type INTERNAL_EVENT. The symbol "$|\!\Rightarrow$" introduces the post-condition-clause; that is, the text appearing after the "$|\!\Rightarrow$" defines the post-condition clause. The result of the post-condition clause is that the parameter e is simply added to the set of internal events. Note that the set of internal events is a dynamic data type, as indicated by the presence of the subscripted "[t]" notation.

Finally, the last form of predicate is called a *schema predicate*. In this case there is both a pre- and post-condition specified. We used such a predicate when we wanted to specify an operation to terminate an acquisition activity (see Eq. (2.32) on page 24). Thus, we declared the following:

Terminate_Acquisition_Activity (a_i: ACTIVITY; t) $\rightarrow$ Boolean
$|\rightarrow$
$\exists$ t' < t • **Initiate_Acquisition_Activity** (a_i: ACTIVITY; t)
$|\Rightarrow$
$\neg$ **Activity_in_Progress** (a_i: ACTIVITY; t)

We read this as follows: The predicate **Terminate_Acquisition_Activity** assumes the value true at a time t, if there is an earlier time t' when the activity was initiated. If so, the predicate denoting **Activity_in_Progress** assumes the value false.

B.5 Expressions

Expressions are statements in *Nestor* that may be constructed from predicates and data types and operations on those quantities. In many cases an expression can involve quantification of a predicate. For example, suppose one wants to specify that a requirement is satisfied. This predicate may be written as

Requirement_Satisfied (r: REQUIREMENT) $\rightarrow$ Boolean

Now suppose there was a set of requirements that was declared as

Requirements: {REQUIREMENT}

The predicate **Requirement_Satisfied** applies to only *one* requirement. If one wants to specify that *every* requirement in the set of requirements was satisfied, this can be specified as

$\forall$ r $\in$ Requirements
Requirement_Satisfied (r: REQUIREMENT)

The above is an example of a simple expression. It is also an example of what is called *promotion*, in that we have taken a predicate that applies to one element of a set and promoted (that is, applied) that predicate over all members of the set.

Earlier, we indicated that predicates can be thought of as the building blocks of a specification. If so, then the expressions can be considered to be the glue that ties the building blocks together. There are many cases where expressions can be applied in the development of a specification. Most importantly, in the framework we use expressions when we develop the notion of timing completeness; see Section 2.5.2 for that discussion.

B.6 Execution Semantics

It is fundamentally important to distinguish between the specification *per se* and an execution of that specification. The latter refers to the operational semantics. We will briefly provide some discussion about this point.

A basic premise in the language *Nestor* (like many formal specification languages) is that the execution of a particular specification statement, such as a predicate, is assumed to be *atomic* in nature. That is, once the statement enters execution, it completes without interruption.

The existence of an atomicity property is especially important in the development of a specification. In our case, it means, for example, that when we developed the specification of the framework we did not have to consider possible interference between operations when they execute. This means, for example, that if the predicate that initiates an acquisition activity were executed, during the time that the predicate executes (i.e., the initiation of an acquisition activity), no other predicate (such as one that might terminate the acquisition activity) is permitted to execute.

The connection between the semantics of a specification and the manner in which that specification could be executed is often subtle. However, imposing an atomicity requirement allows us to develop the specification without reference to the manner in which the specification was executed. In other words, *what* the specification states is different from *how* that specification may be implemented.

Appendix C Guidelines for Model Development

One of the basic reasons for developing the formal specification of an acquisition framework was to allow for the specification of acquisition models. In that sense, an acquisition model is an instance of the framework. Specification of different models allows a more formal comparison of them, as well as gaining a deeper understanding of their semantics and behavior.

In this Appendix, we provide some notes to aid others who may wish to develop a specification of an acquisition model based on the framework defined here. These are intended to provide guidance to the developer; the specification of the waterfall model in Section 3 on page 33 also illustrates the application of a general approach for a simple model. The following discussion is not intended to be a primer on the approach to developing formal specifications; interested readers should consult relevant sources for such broader information.

A key aspect of the development of an acquisition model is identification of the boundary of the scope of the model. This can be more subtle than one might first think. The intended scope of the model determines what activities are performed and other associated properties the model must address, such as relations among activities. In the context of DoD acquisition there is a typical hierarchy of Program Executive Level and Program Office level. But there is also the question of sponsorship. And users. The list can go on; however, the critical point of the scope and boundary of the model must be carefully considered at the outset.

C.1 Conformance to the Framework

Before we discuss model-specific considerations it is worth spending some time examining the notion of conformance to the framework. Recall that one of our goals was to be able to specify the framework such that it could be used in the specification of acquisition models. Loosely speaking, a specification of a model may add new data, operations, and expressions, or refine existing data, operations, and specifications, *provided* that the model does not contradict the framework specification. Some examples of these cases are described below.

With regard to data specified in the framework, the following apply:

- A model cannot change the specification of a framework data element if such a change would violate the framework. For example, in Eq. (2.1) on page 5 we specified that the number of acquisition activities must be greater than zero. Hence, it would be erroneous for an acquisition model to specify a contradiction to this declaration (e.g., that the number of acquisition activities was zero).

- A model may include specification of new data types that are not included in the framework. For example, in Section 5 we discussed possible extensions to the framework that involved data types not defined in the framework, such as entrance and exit criteria.
- A model may refine the specification of data types that are included in the framework. An example of this was discussed in Section 5.8.3 on page 70 where we added detail regarding the structure of an acquisition activity. Note however that dynamic data types declared in the framework (i.e., those with a subscript [t]) may not be converted to static data types and vice versa.

With regard to relations defined in the framework:

- A model may introduce new relations, involving either new data types or data types defined in the framework, provided that they do not contradict relations defined in the framework.
- A model may refine the specification of a relation defined in the framework. Refinement of relations was not used in the development of the framework; an example of where this might be possible would be to state the constraint that a requirement applies to an instance of a system only if the requirement has been approved.[29]

With regard to operations defined in the framework the following apply:

- A model may add new operations that are not present in the framework, provided such operations do not contradict the operations defined in the framework.
- A model may refine the semantics of an operation that is specified in the framework (this is known as operational refinement). For example, the framework includes a predicate **Requirement_Satisfied** that indicates some requirement is satisfied. It does not, however, provide details of the conditions under which the requirement is satisfied. In Section 5.8.2 on page 69 we illustrated an example of how the operation might be refined in the context of a particular model.
- A model is required to resolve the cases in the framework where there are operations whose semantics are underdetermined. There is only one such case in the framework where this arises. In our

[29]. Including constraint clauses in a relation is an advanced feature of the language *Nestor*.

discussion of the semantics associated with suspending and terminating an acquisition activity, the specification of these operations was identical, i.e., underdetermined. A model is required to add additional material to resolve this consideration. See the discussion on this topic following Eq. (2.32) on page 24.

With regard to expressions, the discussion above concerning operations applies. That is, the specification of a model may introduce new expressions, or refine existing expressions, provided that the result does not contradict the specification of the framework. The principal case where expressions are defined in the framework is discussed in connection with model completeness (see Section 2.5 on page 25).

C.2 Basic Data Types

The framework includes a number of basic data types, namely, activities, events, requirements, and instances of a system. Specification of these data types for a particular model is a good starting point. We would suggest that the following questions be resolved:

- What are the basic *activities* that will be included in the model? What is the reason for the choices made? Is it relevant to separate these activities into different sets, such as technical and management activities, or other possible combinations?
- What are the relevant *events* that the model should include? In particular:
  - What are the events that are *internal* to the scope of the project? Typical candidates here include events for reviews.
  - What are the external events that are of interest? Treatment of external events is of special importance because it represents the interaction of the project with the external environment. This may include the need to handle events associated with standards or COTS products, for example.
- How will *requirements* be treated in the model? In particular, do we consider a single set of requirements or, perhaps, multiple, independent sets of requirements? The coupling of requirements, represented by dependency relations (or through specification of predicates) needs to be addressed.
- What approach will be used to specify the *system*? Do we consider a single instance of a system, or do we include multiple instances of a system? The latter question is a classic differentiator between a waterfall model and an incremental model.

Developing approaches to resolve the above questions helps to point us in the direction in which the model development will proceed.

C.3 Relations

Related to the specification of basic data types is the consideration of how the different data types are coupled. This coupling is specified by relations. In the development of the text, we indicated various basic relations that can be formed among the data types.

As a guide to the development of relations it is useful to consider a table, as indicated in the following:

	Activities	**Events**	**Require-ments**	**System Instances**
Activities				
Events				
Requirements				
System Instances				

Table C-1: Developing Relations for a Particular Acquisition Model

Each cell in Table C-1 represents a possible source of relations that can be specified for a particular model. Throughout the text we have used relations to help specify the framework. Many times the relations are developed from a simple diagram such as shown in Figure C-1:

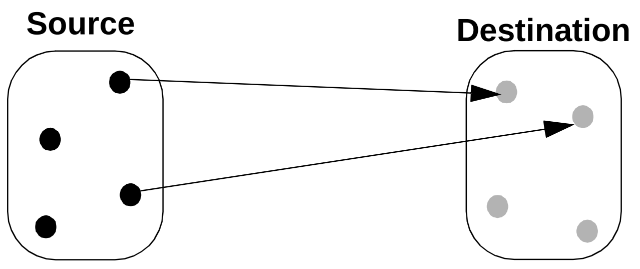


Figure C-1: Simple Model for Development of Relations

A number of items that should be considered when developing relations between a set of source elements and another set of destination elements (which can be the same, of course) are:

- Can multiple source elements be included in the same relation, or does the relation involve only one source element?
- Can multiple destination elements be included in the same relation, or does the relation involve only one destination element?
- Must the relation be defined over all elements in the set of source elements?
- Must the relation be defined over all elements in the set of destination elements?

The above questions are related to identifying the character of a particular relation such as many-to-many or one-to-one. It also involves consideration of whether a relation is total or partial over the set of source and/or destination elements.[30]

C.4 Predicates

Predicates are the main building blocks of the operations that are associated with a specification. Loosely speaking, predicates can be grouped into the following classes:

- axioms: basic statements associated with the model.
- inquiry functions: A predicate can be used to determine the state of some data or operation. For example, in the framework we use inquiry functions (a particular type of predicate) to determine whether an activity is in progress (see Eq. (2.13)) or whether a requirement has been satisfied (see Eq. (2.23)).
- conditional predicates: A conditional predicate is one that has a pre-condition clause that specifies the conditions that must be satisfied in order for the predicate to be true. For example, we used a conditional predicate to specify that in order for an activity to be terminated, it must have been started at some earlier time (see Eq. (2.32)).
- unconditional predicates: An unconditional predicate is one that has only a post-condition clause; no pre-condition clause is present. We used a predicate of this type when we discussed adding a requirement to the set of requirements (see Eq. (2.20)).

30. As an advanced consideration, it is worth considering whether there are constraints involved in the relation. For example, does the relation hold over all elements in the source set or only over some (implicitly defined) subset of the source elements, where the subset is specified by a constraint? For example, one could define a relation between people, with the constraint that it holds only for people under the age of 12.

- schema predicates: A schema predicate is a predicate that has both a pre-condition and a post-condition. The pre-condition clause specified the state that must be satisfied in order to initiate the operation, as described above. The post-condition clause specifies the operations or state that must hold upon completion of the operation specified by the predicate.

As noted, predicates are used to specify operations on state data, such as the requirements associated with the model or operations that can be performed, such as initiating some acquisition activity. Identification of the necessary predicates is largely determined by what is desired to be stated for some aspect of the model.

In addition to predicates, there are expressions that can be developed that assert some property over a specification. For example, the completeness criteria (see Section 2.5) for the framework are presented as expressions. In particular, the criterion that every external event must be associated with some activity (see Eq. (2.38)) is an example of a general expression.

Given the above mechanisms that can be used to develop a specification, some of the questions worthy of consideration include:

- What are the operations that can be applied to the basic data types, such as activities or events? For example, in Section 2.1.1 we included operations that create, modify, and delete a particular acquisition activity. Are these simple operations sufficient for relevant data types, or are more complex operations necessary to develop a model specification?
- What are the operations that couple the different data types? A hint of the relevant operations can be obtained by assessing the relations. As another case, in Eq. (2.27) on page 22 we developed a predicate to indicate that some requirement was satisfied by an instance of a system. This amounts to a coupling of requirements and the system (instance) that manifests those requirements.
- What are the operations that must hold over the entire specification? For example, when we discussed dependencies between requirements, say x and y, we required that for every requirement, if x depended on y, then y did not depend on x (see Eq. (2.25) on page 21). This choice was made to prevent circular dependencies. The fact that this is an assertion over the entire specification is especially important.

Needless to say, the specification of predicates helps to bring out the operational nature of the acquisition model.

C.5 Timing Properties

Timing properties are a crucial element in the specification of the framework and are equally crucial in the development of a model as an instance of the framework. It is consideration of timing properties that leads to a specification of the dynamic character of an acquisition model. In particular, timing properties allow us to specify which activities are serial in nature and which activities can be performed in parallel.

When developing timing properties for a model, some of the relevant issues are:

- What are the criteria for when a given activity may be initiated or terminated?
- What are the criteria for when multiple activities may be performed at the same time? This question is especially important; for example, if one compares a waterfall model with a spiral model. Another example of where parallelism occurs is dealing with external events, such as may be defined for dealing with standards and COTS products. Some work toward this end is discussed in [5].
- Given the possibility of concurrent activities, what is the synchronization between them? For example, if a model specified entrance criteria that depend on multiple concurrent activities, what happens if one activity is complete, but another is not?

There are various timing properties that can be associated with a particular model. It was recognition of this fact that caused us to specify some of these in greater detail. See Section 5.7 on page 63 for some ideas on different timing properties that can be specified.

C.6 Possible Extensions

The intent of the framework specified in this report was that it be of sufficient breadth that it can be used to develop multiple acquisition models. At the same time, however, we have tried not to overly constrain the specification. It was the need for balance that led us to consider possible additional items that one could include in the specification of a model. These possible extensions were discussed in Section 5.

The relation between the extensions and the basic framework specification is shown in Figure C-2. The center of the figure shows the (core) framework specification. It is surrounded by possible extensions, many of which were described in Section 5. Another item that has been mentioned in the text is how an acquisition model would need to account for standards and COTS products. But also shown in Figure C-2 is an indication that other topics can be defined as part of developing a specification of an acquisition model. The significance of the arrows leading to the framework simply indicates that the extensions are integrated into the basic specification of the framework.

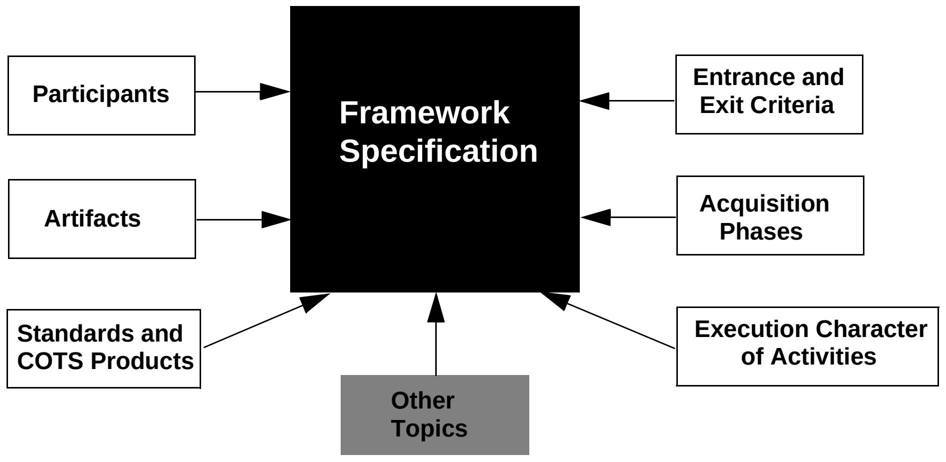

Participants
Framework Specification
Entrance and Exit Criteria
Artifacts
Acquisition Phases
Standards and COTS Products
Execution Character of Activities
Other Topics

Figure C-2: Relation of Extensions to Framework Specification

It is difficult to give specific guidance on additional specification that could be included in a particular model. Recognition of the need to state some property associated with the model is what will determine whether additional specification is required that is outside the scope of the core framework. Thus, treatment of extensions is very model dependent.

C.7 Assertions Over the Specification

The ultimate utility of a formal approach to the specification of an acquisition model is that it should be possible to reason about the specification in a formal manner. From a mathematical point of view, this deals with the development of assertions which can then be established by a proof technique.

The ability to state, and then prove, assertions about a formal specification is very powerful. In the discussion of the waterfall model (Section 3.5 on page 43) we gave some instances where it was possible to reason about a model specification. The ability to pose questions about a model, and then to be able to resolve the questions through application of a formal approach, is very useful. While a discussion of proof techniques is beyond the scope of this report, we want to make the reader aware that such approaches do exist and are of considerable value.

Appendix D Additional Comments on the Waterfall Model

In Section 3 we provided a specification of the traditional waterfall model in terms of the basic acquisition framework described in Appendix D. There is additional material in reference [6] concerning the waterfall model that we did not address earlier. In particular, the additional specification statements concerning the waterfall model can be viewed as refinement to the basic specification. Our goal here is to address the refinement issues for two topics, namely, elaboration of activities that are included in the model and a specification of artifacts associated with the model.

D.1 Elaboration of Activities

The activities described for the waterfall are specified at a high level. It is possible to refine those activities, which amounts to the addition of detail to a specification. In the case of the waterfall model, the activity of preliminary design included the following activities:

- Document system overview.
- Design database and processors.
- Allocate subroutine storage.
- Allocate subroutine execution times.
- Describe operating procedures.

We can declare the above items as activities as

```
Document_System_Overview, Design_Database, Allocate_Subroutine_Storage,
Allocate_Subroutine_Execution_Time, Describe_Operating_Procedures:
    ACTIVITY
```

Next, we declare the above to be members of a set of design activities:

```
Design_Activities = {Document_System_Overview, Design_Database,
    Allocate_Subroutine_Storage, Allocate_Subroutine_Execution_Time,
    Describe_Operating_Procedures}
```

It is now possible to relate the above activities to those defined in Section 3. In particular, we earlier defined an activity for program design:

Design_Activities ⊃ Program_Design

D.2 Artifacts

The description of the waterfall model also included a number of artifacts that are created during the development process. Specification of artifacts is not part of our core framework; however, in Section 5.4 we described an approach to dealing with artifacts.

We begin with a description of a free type to represent an artifact:

[ARTIFACT]

In the description of the waterfall model provided in [6], a number of artifacts are described. These are documents that are produced during the development effort. We can declare these to be artifacts as follows:

Software_Requirements_Document, Preliminary_Design_Specification,
Interface_Design_Specification, Final_Design_Specification, Test_Plan_Specification,
Operating_Instructions: ARTIFACT

We now declare a set of artifacts that we will denote as documents:

Documents: {ARTIFACT}

Documents: Software_Requirements_Document, Preliminary_Design_Specification,
Interface_Design_Specification, Final_Design_Specification, Test_Plan_Specification,
Operating_Instructions

It is also necessary to relate the documents to a particular activity in the model. The character of the relation is that every document must be related to at least one activity, but not all activities need to have an associated document. The specification of the necessary relation is then

Document_Relation: Documents <----|-> Activities
$\exists\ d_i \in$ Documents, $a_j \in$ Activities
• **Document_Relation** (d_i, a_j)

A declaration of the relations for the above documents with their associated activities are defined as:

Document_Relation (Software_Requirements_Document, Software_Requirements);
Document_Relation (Preliminary_Design_Specification, Program_Design);
Document_Relation (Interface_Design_Specification, Program_Design)

Document_Relation (Final_Design_Specification, Program_Design)
Document_Relation (Test_Plan_Specification, Testing)
Document_Relation (Operating_Instructions, Operations)

Appendix E Index of Mathematical Formulas

The following index references data types and operations that appear in this report. Page numbers appearing in bold font are associated with the basic framework.

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.		
6. AGENCY USE ONLY (leave blank)	7. REPORT DATE December 2001	8. REPORT TYPE AND DATES COVERED Final
9. TITLE AND SUBTITLE A Framework for the Specification of Acquisition Models		10. FUNDING NUMBERS C — F19628-00-C-0003
11. AUTHOR(S) B. Craig Meyers and Patricia Oberndorf		
12. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		13. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2001-TR-004
14. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		15. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-2001-004
16. SUPPLEMENTARY NOTES		
12.a DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12.b DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

This report describes a framework for the specification of acquisition models. The exposition is formal in nature. The framework is defined in terms of activities, events, requirements, and instances of a system. In addition, various relations among these items, such as the relation between acquisition activities and acquisition events, are defined. The timing properties associated with the items receives special treatment.

The value of a framework is that one can develop specifications of various acquisition models, such as waterfall, spiral, or incremental, as instances of that framework. Formalizing the specification of an acquisition model has benefit in that one can reason about the characteristics of the domain addressed by the model. When this is done for multiple acquisition models, each derived from the same framework, it is possible to compare different acquisition approaches.

14. SUBJECT TERMS acquisition; acquisition process; formal model			15. NUMBER OF PAGES 132
			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

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102