Possible UML-Based Size Measures

Richard D. Stutzke
Science Applications International Corp.
6725 Odyssey Drive
Huntsville, AL 35806-3301
(256) 971-6224 (office)
(256) 971-6550 (facsimile)
Richard.D.Stutzke@cpmx.saic.com

07 October 1998

© 1998 by Richard D. Stutzke

ABSTRACT

Why do we size software? For many reasons. To estimate effort. To estimate memory requirements. To estimate processing or execution speed. This paper considers size measures that help us predict the effort needed to design, build and test computer software using object-oriented methods. The Unified Modeling Language (UML) is emerging as an industry standard for recording the results of object-oriented analysis and design. We focus specifically on the use of the products of the UML as early reliable measures of software size. These products include use cases, class diagrams, state diagrams, event trace diagrams, etc. There are too many UML products to use them all. Thus, we identified a subset suitable for early estimation of effort by examining the development process. The process determines the activities performed by the development staff, and so the sequence in which the products are generated. This enabled us to identify which products will be available early in a project.

We estimate the size by combining estimates for the three aspects of software: data, control behavior, and function, as identified by [DeMarco, 1982]. We examined the elements of UML notation, and the activities of the Object-Oriented Systems Analysis (OOSA) method. This method is similar to Object Management Technique (OMT) which has been absorbed into UML. The Class Diagram (essentially an Entity Relationship Diagram) and the Event List or, better, the Statechart (a State Transition Diagram) seem to suffice to determine most of the development effort. The only activity we think is not covered is the development of complex operations (algorithms, methods). We recommend an approach to help the estimator identify these and estimate the additional effort. We attempted to estimate the size in Unadjusted Feature Points. We also surveyed other relevant approaches mentioned in the literature. We conclude by identifying some questions that need to be addressed in the future.
The design goals of UML are to:

- Provide an expressive visual model language
- Integrate best OO practices
- Be independent of programming language
- Be extensible and customizable

Analysts and developers can use UML to specify, visualize, and document models of problems, and systems. They can also use it to construct code (possibly even generating portions automatically).

The first version of UML was released in 1996. Since then the definition has been refined. Version 1.1, dated 01 September 1997, identifies roughly two dozen diagrams, grouped into the following categories:

- Diagram Elements
- Static Structure
- Use Case Diagrams
- Sequence Diagrams
- Collaboration Diagrams
- Statechart Diagrams
- Activity Diagrams
- Implementation Diagrams
- Model Management and Extension Mechanisms

Diagram elements describe the notation used by all of the other diagrams. The next six items are used to capture the results of analysis. Implementation Diagrams record details associated with the physical design. The Model Management diagrams are used to organize the other diagrams, providing a roadmap to help readers navigate through the many diagrams which comprise the model. We are interested in the analysis diagrams as sources of size information.

2.0 Defining a Size Measure

Our goals are to define a size measure that:

- Is correlated to the development effort
- Is independent of technology and production process
- Can be estimated early in the project
- Can be estimated easily

These goals are similar to the goals behind the development of Function Point Analysis [Albrecht, 1979]. Also see [Albrecht, 1984]. [Jones, 1991] provides a good summary of the various types of Function Points and their history. [Behrens, 1983] describes the use of Function Points to measure development productivity.
1.0 Introduction

Many developers presently use object-oriented methods to design, build and test computer software. We thus desire to identify size measures that we can use to estimate the development effort early in a project. The Unified Modeling Language (UML) is emerging as an industry standard for recording the results of object-oriented analysis and design. We thus focus specifically on the use of the products of the UML as early reliable measures of software size.

We begin by describing our motivation. Then we review the products of UML that are potential candidates. We explain our strategy for selecting the most suitable products, and propose specific measure of size based on characteristics of these products. We also identify related approaches proposed by other authors. We conclude by identifying topics for future study.

1.1 Object-Oriented Development

Programmers now agree that the best way to package software components is as objects which combine data, functions and (optionally) state (memory of previous events). The general concept is to encapsulate data and the functions that manipulate the data. Objects have clear boundaries, allowing developers to connect them together easily (using inheritance and message passing). These software objects are called classes. Libraries of classes are available and programmers assemble products by selecting classes, and integrating them with new custom-built classes. Intelligent tools assist the developer in assembling and testing the products. Objects correspond to entities in the real world such as physical objects, roles, and events. Various analysis methods help developers identify objects relevant to the application and solution domains. These methods use graphical notation, supplemented with text, to record various features of the objects (classes). The products of a method describe data, function, and state. (I use "behavior" to mean control and sequencing of operations. Standard object-oriented terminology uses the word "behavior" to refer to operation, methods, and functions, and "state" to refer to the control aspects. Actually, as we will see, UML records the operations in the boxes in the statechart so the two are closely related.)

Analysis and design progress through three steps. Logical design describes the problem and the required capabilities. Physical design extends the logical design by adding details of the specific implementation chosen. The production process converts the physical design into software objects and ultimately into working code, and then tests the product. Software development methods ("methodologies") endeavor to allow a smooth transition from analysis to design to implementation. (The Shlaer-Mellor method has the goal of eliminating coding entirely. Implementation reduces to running a code generator and then testing the product.) This progression is the basis for our strategy to define a size measure. (See Section 2.2.)

1.2 Unified Modeling Language

Unified Modeling Language (UML) endeavors to bring together the best of several Object-Oriented (OO) modeling approaches. The main authors are Grady Booch (developer of the Booch Method), James Rumbaugh (co-developer of the Object Management Technique, OMT), and Ivar Jacobson (inventor of use cases). The Rational Corporation has sponsored much of the work. (See the Rational Web site for additional information.)
2.1 Strategy

Our strategy to achieve these goals is to:

- Base the size measure on the products of logical design
- Hide the effects implementation and production in the productivity
- Use only UML elements produced during analysis
- Use a subset of these UML elements

UML products include use cases, class diagrams, state diagrams, event trace diagrams, etc. There are too many UML products to use them all. In addition, there is some overlap in the information contained in the various diagrams. Thus, we needed to identify a subset suitable for early estimation of effort.

We used DeMarco's three perspectives required to completely specify a software system [DeMarco, 1982, page 64]:

- Retained Data (content and structure)
- Function (algorithms)
- Behavior (state transition)

We compare this to the facets of an object:

- Attributes (data)
- Behavior (operations, methods, functions)
- State (memory)
- Identity (a key - a data attribute)
- Responsibilities ("purpose", raison d'etre)

We think that DeMarco's three perspectives are a good way to partition our size measure and can be mapped to the facets of objects as captured using UML diagrams. DeMarco's perspectives are also used by [Whitmire, 1996] to define 3D Function Points. (We will discuss this later.) One issue is how to combine the three size components into a single measure of size. (Note that DeMarco identifies two measures called Function Bang and Data Bang. He says (page 89) to keep them separate. The reason is that some project activities are primarily concerned with implementing functions, while other are concerned with implementing data. Thus, the estimated effort for these activities should depend on Function Bang or Data Bang, respectively. In the object-oriented world, however, data and function are encapsulated together and so it makes sense to try to combine them.) Following Whitmire, we will try to define the size of each component in Feature Points so the values can simply be added. See [Jones, 1991] for a description of the various types of Function Points and Feature Points.

2.2 Modeling Process

The modeling process determines the activities performed by the development staff, and so the sequence in which the products are generated. This enables us to identify which products will be available early in the project. The process is guided by a method. There are many methods
available. (UML is a notation to record the results of an analysis method but does not prescribe how to produce the products, nor the order in which they are produced. UML may evolve to include such information.)

The Shlaer-Mellor method [Shlaer, 1982] and [Shlaer, 1992] is considered by many developers to be especially good for analysis. We thus choose it to identify the products and order in which the products are generated. The Shlaer-Mellor method generates products in the following order:

1. Object (class) diagram and Data Dictionary
2. Event Dictionary and Event Scenarios
3. State Diagram (for each object or class)
4. Data Flow Diagram (for each state)
5. Event Trace Diagram (links events and objects for each scenario)
6. Class Interaction Diagram
7. Context Diagram

Items 5-7 are actually started early and then revised. 'They help organize the analysis process and manage the evolving model.' The analyst moves from the object view (identifying the classes, that is the data), to the dynamic view (identifying events and state diagrams), and then to the functional view (data flow and transformation, that is methods and operations).

2.3 UML Components for Data, Function, and Control

Tables 1, 2, and 3 list the components of UML that we think represent data, control (states), and functions. Note that some diagrams are listed in more than one table because the different perspectives are mixed in those diagrams.

---

Actually, there is a great deal of iteration during analysis. Information from the later products is used to cross check, correct, and refine the products produced earlier. This is why some means of estimating the growth in the number of classes, states, etc. over time will be needed. This is a topic for future research.

Possible UML-Based Size Measures 5 08SEP08
### Table 1 UML Diagrams Which Describe Data

<table>
<thead>
<tr>
<th>UML COMPONENTS FOR DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class diagram shows:</strong></td>
</tr>
<tr>
<td>- Class</td>
</tr>
<tr>
<td>- Class attributes</td>
</tr>
<tr>
<td>- Relations/associations (multiplicity and conditionality [obligation] only. Ignores fidelity.)</td>
</tr>
<tr>
<td>- Aggregation and composition</td>
</tr>
<tr>
<td>- Generalization and specialization</td>
</tr>
<tr>
<td><strong>Collaboration diagram shows:</strong></td>
</tr>
<tr>
<td>- Collaboration (communication)</td>
</tr>
<tr>
<td>- Message syntax</td>
</tr>
<tr>
<td><strong>Sequence diagram shows:</strong></td>
</tr>
<tr>
<td>- Flow of messages between objects in time sequence (event trace diagrams, scenarios)</td>
</tr>
<tr>
<td><strong>Use case diagram shows:</strong></td>
</tr>
<tr>
<td>- Collaboration between system and actors in the environment</td>
</tr>
<tr>
<td>- Relations between use cases</td>
</tr>
</tbody>
</table>

### Table 2 UML Diagrams Which Describe Functions

<table>
<thead>
<tr>
<th>UML COMPONENTS FOR FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence diagram shows:</strong></td>
</tr>
<tr>
<td>- Flow of messages between objects in time sequence (event trace diagrams, scenarios)</td>
</tr>
<tr>
<td><strong>Use case diagram shows:</strong></td>
</tr>
<tr>
<td>- Collaboration between system and actors in the environment</td>
</tr>
<tr>
<td>- Relations between use cases</td>
</tr>
<tr>
<td><strong>Collaboration diagram shows:</strong></td>
</tr>
<tr>
<td>- Collaboration (communication)</td>
</tr>
<tr>
<td>- Message syntax</td>
</tr>
</tbody>
</table>
Table 3 UML Diagrams Which Describe Behavior

<table>
<thead>
<tr>
<th>UML COMPONENTS FOR CONTROL</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Statechart shows:</td>
</tr>
<tr>
<td>- Sequence of states for a reactive class</td>
</tr>
<tr>
<td>- Stimuli and response(s) thereto</td>
</tr>
<tr>
<td>- Includes nested, sequential, and orthogonal states (orthogonal = independent, nonconcurrent)</td>
</tr>
<tr>
<td>- Activity diagram shows:</td>
</tr>
<tr>
<td>- Transitions taken when a state activity is completed</td>
</tr>
<tr>
<td>- A special case of statecharts</td>
</tr>
<tr>
<td>- Good for forks and joins (rendezvous, task synchronization)</td>
</tr>
<tr>
<td>- Implementation diagrams show:</td>
</tr>
<tr>
<td>- Run-time dependencies of components (e.g., rendezvous, task synchronization)</td>
</tr>
<tr>
<td>- Packaging structure of components of the deployed system (allocation of components to nodes, physical topology)</td>
</tr>
</tbody>
</table>

2.4 Historical Perspective

Many authors have provided size measures that address one or more of DeMarco's three perspectives. Table 4 lists some of these, and shows which perspective(s) each addresses. The most complete appears to be 3D Function Points (3DFPs) defined by Whitmire [Whitmire, 1995] which is similar to our approach.

Predictive Object Points (POPS) [Minkiewicz, 1997] appears to be based on the Metrics Suite for Object-Oriented System Environments (MOOSE) defined by [Chidamber, 1994]. POPS are similar to 3DFPs in that they address three axes of OO system complexity: methods and data, inheritance, and interobject communication. POPS are analogous to Function Points but the author claims their definition is based on behavior and state. Closer analysis reveals, however, that the constituent measures are: the weighted measures per class (WMC), the number of base (top level) classes (TLC), the number of children (NOC), and the average depth of a derived class in the inheritance tree (DIT). No measure relating to the number of states appears. In our opinion, POPS seems to be programmer-centric instead of customer-centric. It is basically a structural decomposition that reflects some implementation decisions (via NOC and DIT). This also means that the estimator must analyze the application in greater detail to obtain the necessary measures, delaying the time when the first estimate can be made. For these reasons we did not pursue POPS.

The MOOSE metrics [Chidamber, 1994] attempt to map directly to the first three steps of the Booch method: identify classes and objects, identify semantics of classes and objects, and identify relationships between classes and objects. MOOSE uses six measures which can be determined during the early design process: the weighted measures per class (WMC), the average depth of a derived class in the inheritance tree (DIT), the number of children (NOC), the coupling between objects (CBO), the response for a class (RFC), and the lack of cohesion of methods (LCOM). Unlike POPS, the number of base (top level) classes does not appear. States do not appear explicitly but perhaps RFC includes some state-like aspects. (This needs to be confirmed
once we receive the original paper.) We did not pursue MOOSE further for the same reasons we did not pursue POPS (no states, low [design] level measures).

Object Point Analysis (OPA) [Gupta, 1996] identifies objects and assesses their individual complexity. The complexity depends on the object's effective attributes (EA), instance connections (IC), processing complexity (PC), and message connections (MC). The estimator uses a table to convert these ratings into numerical values that are used to compute the Base Object Points (BOPs). These correspond to Unadjusted Function Points. Then the estimator computes an Adjustment Factor (AF) based on 14 system characteristics which are an "evolutionary version" of the original FPA set. OPA does not seem, however, to directly address states. (This needs to be confirmed once we receive the original paper.) There was not time to analyze OPA in detail to determine if a suitable mapping to UML diagrams is possible. The apparent lack of states, however, appears to us to be a serious shortcoming. We did not pursue OPA for this reason.

Table 4 Previous Size Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Data</th>
<th>Function</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Bang [DeMarco, 1982]</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Data Bang [DeMarco, 1982]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feature Points [Jones, 1988]</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Mark II Function Points [Symons, 1988],</td>
<td>x</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>[Douglas, 1995]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSET Function Points [Reifer, 1990]</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Moose [Chidamber, 1995]</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Object Points [Boehm, 1995]</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>3D Function Points [Whitmire, 1995]</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>FPA and Use Cases [AMS, 1996]</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPA [Gupta, 1996]</td>
<td>x</td>
<td>x</td>
<td>?</td>
</tr>
<tr>
<td>Predictive Object Points [Minkiewicz, 1997]</td>
<td>x</td>
<td>x</td>
<td>?</td>
</tr>
</tbody>
</table>
3.0 The Size Measures

We define a size measure for each of DeMarco's three perspectives in this section. We then combine them to obtain the total size. We also mention the relation of our size measure to Whitmire's 3D Function Points. The section concludes with some data on effort allocation published in [Montrose, 1996].

3.1 Data Size

The Class Diagram is essentially an Entity Relationship Diagram. DeMarco computes Data Bang by assigning a weight based on the number of relations an entity (class) has and then sums these over all entities [DeMarco, 1982, pages 89-90]. He uses these weights:

<table>
<thead>
<tr>
<th>( R_i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( W(R_i) )</td>
<td>1.0</td>
<td>2.3</td>
<td>4.0</td>
<td>5.8</td>
<td>7.8</td>
<td>9.8</td>
</tr>
</tbody>
</table>

This model ignores the internal data content of the entity. We decided that we needed to include a measure of the number of attributes.

3.1.1 Douglas and Treble Model

Neil Douglas and Steve Treble describe a technique for estimating function points from a data model in [Douglas, 1995]. Their technique is applied to Mark II Function Points defined in [Symons, 1991], based on early work reported in [Symons, 1988]. They note that a similar technique could be applied to IFPUG Function Points [IFPUG, 1990] but this will be more difficult since IFPUG's method groups entities into logical files. We chose their approach as the basis to estimate data size.

Douglas and Treble develop an estimating relation by making a number of assumptions. This derivation has eight steps and produces Mark II function points. The steps are:

1. Size the components of the data model.
2. Estimate the number of transactions.
3. Estimate the average number of entities accessed per transaction.
4. Estimate the number of fields per transaction.
5. Estimate the size of update and create transactions.
6. Estimate the size of query transactions.
7. Estimate the size of delete transactions.
8. Sum the sizes to get total system size in unadjusted function points.

The next few paragraphs summarize their derivation.

Step 1. Using the data model, estimate the number of entities, the number of attributes on each entity and the number of relationships between entities (i.e. the number of lines, or relations, connecting entities together). This gives:

\[ N_e = \text{Number of Entities} \]
\(N_s\) = Number of Attributes
\(N_r\) = Number of Relations

Step 2. Estimate the number of transactions for each entity. They assume that each entity in the system will be created, changed, deleted, and queried once. This gives four transactions for each entity:
\(N_t\) = Number of transactions = \(4N_e\)

Step 3. Estimate the average number of entities accessed per transaction. They define the average connectivity (\(A_c\)) as the number of entities connected to a particular entity. Then the average number of entities accessed per transaction will be the sum of the entity on which the transaction is based plus \(A_c\) others:
\(N_{et} = \text{Number of entities per transaction} = 1 + A_c\)

They estimate the average connectivity as:
\(A_c = (2N_e)/N_e\)

where the factor of 2 comes from assuming that each relation really goes two ways (A's relation to B and B's relation to A) and so gives rise to two transactions.

Step 4. Estimate the number of fields per transaction. They assume that all the attributes of the major entity plus half of the attributes of the connected entities will be involved in a transaction. This gives the number of fields involved in a transaction:
\(N_{ft} = \text{Number of fields per transaction} = (N_s/N_e) + A_c(N_s/N_e)/2 = (N_s/N_e)(1 + N_s/N_e)\)

They divide by \(N_e\) here because the numbers \(N_e, N_s,\) and \(N_r\) are the total for the entire system. The calculation of \(N_{ft}\) is PER ENTITY.

The remaining steps are based on the industry standard weights used to calculate unadjusted function points, i.e. input fields are weighted by 0.58, processed entities by 1.66 and output fields by 0.26.

Step 5. Estimate the size of update and create transactions. They assume that the largest part of the dataflow in an update or create transaction will be input with a nominal output flow, and so the average size for an update or create transaction is:
\(0.58*N_{ft} + 1.66*N_{et} + 0.26\)

They assume that half of all transactions will be update or create transactions. These assumptions give:
\(SUCT = \text{Size of update and create transactions} = (0.58*N_{ft} + 1.66*N_{et} + 0.26)*N_e/2\)
Step 6. Estimate the size of query transactions. They assume that the largest part of the data flow for query transactions will be output with a nominal input flow, and so the average size for a query transaction is:

$$0.58 + 1.66*{\text{Net}} + 0.26*{\text{Nt}}$$

They assume that one-fourth of all transactions will be query transactions. These assumptions give:

$$\text{SET} = \text{Size of enquiry transactions}$$
$$= (0.58 + 1.66*{\text{Net}} + 0.26*{\text{Nt}})*{\text{Nt}}/4$$

Step 7. Estimate the size of delete transactions. They assume that delete transactions are likely to be processing heavy with nominal input and output data flows, and so the average size for a delete transaction is:

$$0.58 + 1.66*{\text{Net}} + 0.26$$

They assume that one-fourth of all transactions will be delete transactions. These assumptions give:

$$\text{SDT} = \text{Size of delete transactions}$$
$$= (0.58 + 1.66*{\text{Net}} + 0.26)*{\text{Nt}}/4$$

Step 8. Compute the total system size in unadjusted function points by summing the results of steps 5, 6, and 7:

$$\text{UFPs} = \text{SUCT} + \text{SET} + \text{SDT}$$

This result can be simplified by combining the relevant portions of the terms, expanding them in terms of Na, Nr, and Ne, and then simplifying the resultant expressions. This gives:

$$\text{UFPs} = \text{Tif}*0.58 + \text{Tea}*1.66 + \text{Tof}*0.26$$

where:

$$\text{Tif} = \text{Total input fields} = 2{\text{Ne}}*(1 + {\text{Nt}}/{\text{Ne}}) + 2{\text{Ne}}$$
$$\text{Tea} = \text{Total entities accessed} = 4{\text{Ne}} + 8{\text{Nr}}$$
$$\text{Tof} = \text{Total output fields} = {\text{Na}}*(1 + {\text{Nt}}/{\text{Ne}}) + 3{\text{Ne}}$$

Expanding and collecting terms gives:

$$\text{UFPs} = 1.42*({\text{Na}} + 6.0*{\text{Ne}} + 9.4*{\text{Nr}} + {\text{Na}}*{\text{Nt}}/{\text{Ne}})$$

Based on this result, the model of Douglas and Treble indicates attributes are "cheap" (relative weight of 1 versus 6.0 or 9.4). It also indicates the relations "cost" about 56% more than entities. Lastly, the term "Na*Nt/Ne" may be a candidate for a measure of entity complexity.

We will define the size for the data component of the objects to be:

$$\text{Size(data)} = \text{UFPs} = 1.5*({\text{Na}} + 6*{\text{Ne}} + 9*{\text{Nr}} + {\text{Na}}*{\text{Nt}}/{\text{Ne}})$$
3.1.2 Functional Content of Relations

Relations have certain characteristics which affect the amount of functionality the developers must implement. Each relation is characterized by Multiplicity, Conditionality, and Fidelity. Multiplicity indicates how many entities are connected together by an instance of the relation. Conditionality indicates if a relation always exists or if it can be created and destroyed. (Some authors call this the "obligation" of the entity to participate in the relation.) Fidelity indicates how many such relations the entity can participate in at one time. (The values are monogamous or polygamous.) UML notation does not presently include fidelity.

The static structure ("data") is characterized by the parameters $N_e$, $N_a$, and $N_r$. These seem adequate to estimate the data-related part of the size and so the associated development effort. The amount of functionality that must be implemented depends on the number of policies that need to be enforced. A plausible model is to add effort proportional to the number of policies (Conditionality, Multiplicity, Fidelity). If we assume 30 Source Lines Of Code (SLOC) to implement the logic for one policy, and also that one function point is about 100 SLOC, then each policy contributes 0.3 Function Points. This is small compared to the size of an Internal Logical file (10 IFPUG Function Points or 7 Feature Points) and seems reasonable as a first guess.

3.2 State Size

Active objects exhibit interesting behavior (both states and functions). Such objects typically:

- Perform actions autonomously
- Coordinate the activities of component objects
- Generate events

Active objects produce or analyze data ("functions") or produce or control actions ("behavior").

Each active object has a state model (represented in UML by a Statechart). They may also have an Action Specification (represented in UML by the pseudocode in the boxes of the Statechart, or via scenario diagrams or supplemental notes).

The Statechart (a form of State Transition Diagram) seems to be the best choice to gauge the size related to the states. The Event List is another possibility. We thus propose the following two approaches.

First, we can count the states and transitions directly from the Statechart. This is analogous to the approach used for the Class Diagram (Entity Relation Diagram). We only count active objects (since only they have states.) Let:

$N_s = \text{Number of states}$

$N_t = \text{Number of transitions}$

The number of states is the total of all "bubbles" on all Statechart diagrams in the model. The number of transitions is the total number of "lines" on all diagrams in the model. Note that some lines arise from the Object Communication Model which shows the flow of messages (events)
between objects. Assuming that states and transitions take the same effort to define (i.e., one function point each) gives:

\[
\text{Size}(\text{states}) = N_s + N_t
\]

Second, we could just count the number of events in the Event Dictionary (Event List), \(N_{ed}\). If we assume one event per state, and one feature point per state, we have:

\[
\text{Size}(\text{state}) = N_{ed}
\]

We prefer the first approach since it has two parameters and so, hopefully, will be more accurate. We may still need to adjust the weighting factors in these size calculations to correctly normalize the contribution of the states to the total size. (See Section 3.5.)

3.3 Function Size

The Statechart contains descriptions of the operations performed in each state. These descriptions resemble pseudocode or process specifications (p-specs) used in structured analysis methods. These descriptions suffice to define the operations (functions) in most cases because oo functions are decomposed to a low level and so tend to be simple. Complicated descriptions are not needed.

Some systems will, however, need more complex operations (algorithms, methods). We need a way to estimate the effort to develop these "hard" operations. (This is similar to the inclusion of "algorithms" in Feature Points.) We propose to use COCOMO's definition of the complexity cost driver to help the estimator identify these "hard" operations. See [Boehm, 1981] for the original definition of this cost driver. [Boehm, 1995] extends and updates the original definition. This can be done as follows. The analyst records a list of candidate algorithms as work proceeds on the Data (Class) and State models. Once the Class Diagram and Statechart are done, the analyst can assign a complexity rating to each candidate algorithm using the COCOMO definition for complexity. (We rate each dimension of complexity: control, computation, device handling, data management, user interface, and then compute a (weighted) sum to get the rating.) We can then assign a size at least two ways. First, we can add a fixed size (say 3 Feature Points) for each algorithm with a complexity above some threshold. Second, we can multiply the size (3 Feature Points) by COCOMO's complexity cost driver.

We do not think that "hard" operations will contribute much to the total size. As already noted, most of the operations are simple, and are included in the data and state sizes. Since there are few "hard" operations, the method to estimate the function (algorithm) size is not too critical.

3.4 Total Size

We estimate the size by combining estimates for the three aspects of software: data, control behavior, and function, as identified by [DeMarco, 1982]. Table 5 lists the elements of UML that we use to estimate the size. The table shows the adjustments for entity "complexity" and "relation management". There seems to be no obvious measure of "behavioral complexity". Complicated operations are identified using COCOMO's criteria for the Complexity cost driver.
Table 5 Summary: Elements to Use

SUMMARY: ELEMENTS TO USE

- Class diagram (OIM, ERD)
  - Number of entities
  - Number of attributes
  - Number of relations
  - Adjust for entity “complexity” \((N_e*N_e/N_c)\)
  - Adjust for “relation management” (policies enforced)

- State Charts
  - Number of states
  - Number of transitions
  - No apparent measure of “behavioral complexity”

- Analysis Notes
  - Identifies “complicated operations” (3 feature points each)
  - COCOMO’s complexity ratings perhaps usable

The total size is the weighted sum of:

\[
\text{Size(total)} = W_d*\text{Size(data)} + W_s*\text{Size(state)} + W_f*\text{Size(function)}
\]

We attempted to estimate the size in Unadjusted Feature Points so we can just sum the three values. (The weights are unity.) More complex functional forms are not justified given the lack of empirical data on the effort expended for OO projects.

3.5 Relation to 3D Function Points

Of the other relevant approaches mentioned in the literature, only Whitmire’s addresses all three perspectives. We thus compare his 3D Function Points and our proposed size model.

Whitmire’s data size is determined using standard Function Point analysis [IFPUG, 1994]. The value is based on the number of Data Element Types (DETs), Record Element Types (RETs), and File Types Referenced (FTRs). The values of pairs of types are used in lookup tables to obtain the size of the entity.

Whitmire defines transformations as processes that convert a set of input data into a set of output data. Transformation alter the fundamental nature or content of the data, or create new data. He thus excludes loading data into files, and formatting data for output since the data is essentially unchanged. Transformations are derived directly from the problem space. Each operation is treated as a unique transformation.

He computes the size of a transformation based on the number of steps and the number of semantic statements which constrain the calculation. (Semantic statements are predicates that must remain invariant throughout the sequence of operations.) Transformations are assigned values of 7, 10, and 15 Function Points for Low, Average, and High complexity, respectively.

He originally (1992) computed control size based on states and transitions taken directly from a Finite State Machine representation. (State Transition Diagrams are one way to depict
these.) He simply summed the counts of both states and transitions. He also collapses multiple transitions between a pair of states into a single transition for counting purposes. His later experience (1995) with state modeling caused him to count only the transitions. He cites two reasons for this change. First, we take many states into account when we count the semantic statements for a transformation. (A precondition for an operation is just another way of saying that in order for the operation to fire, the application must be in a given state.) Second, because transitions are used to model stimulus/response or event/action pairs, the number of transitions is directly determined by the application domain. This leads Whitmire to measure the control dimension by counting only the transitions in the state model. Transitions are not assigned a level of complexity. He assigns each transition a value of 3 Function Points. (Based on Whitmire’s rationale, our computation of the state size may need to be revised. At least, the weights assigned to states and transitions should be increased by 50% or so.)

Whitmire computes the total size by summing the counts for inputs, outputs, queries, internal files, external files, transformations, and transitions. Whitmire's approach to counting states is essentially the same as ours. He counts transformations differently, and we believe that our approach is more reasonable for Object-Oriented software since our approach naturally seems to encompass the encapsulation of data and functions within the classes.


3.6 Computing Effort

Once we have estimated the total size as described above, we estimate the total effort by dividing the size by the productivity (measured in Feature Points per person-hour). The productivity must account for characteristics of the physical design (architecture, platform, etc.) and of the development process (knowledge, skill, tool, reuse of existing components, etc.). We suggest that COCOMO's cost drivers for product performance, project constraints, and personnel abilities be used as a basis for a productivity model. Possibly the OO size can be used in lieu of Unadjusted Function Points in the COCOMO II model.

3.7 OOSA/RD Effort Data

Once we have the total effort, we will want to allocate it to various activities. We include here an example from [Montrose, 1996]. He provides data from a project which built an embedded product written in C++ with over 200 classes. The team used Shlaer-Mellor's Object Oriented Systems Analysis and Recursive Design Method (OOSA/RD). Table 6 shows the breakdown of the engineering effort by different activities. Model development consumed 65% of the effort. (Montrose claims that they observe a range of 50-75% for this.) Table 7 shows a breakdown of the modeling effort (the 65%). Montrose stated that administrative costs added 14% to the total effort shown in Table 6.
Table 6 Breakdown of Effort Expended for Different OOA/RD Activities

<table>
<thead>
<tr>
<th>Activity</th>
<th>Effort (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition domains, build Domain chart</td>
<td>2%</td>
</tr>
<tr>
<td>Build, review Object Information Models</td>
<td>21%</td>
</tr>
<tr>
<td>Build, review State Models</td>
<td>305</td>
</tr>
<tr>
<td>Build, review Action Specification</td>
<td>12%</td>
</tr>
<tr>
<td><strong>Total OOA time</strong></td>
<td><strong>65%</strong></td>
</tr>
<tr>
<td>Develop and test Software Architecture and Translation rules</td>
<td>10%</td>
</tr>
<tr>
<td>Manually translate OOA models into code, integrate and test</td>
<td>25%</td>
</tr>
</tbody>
</table>

Table 7 Breakdown of the OOA/RD Modeling Effort

<table>
<thead>
<tr>
<th>Activity</th>
<th>Effort (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td>77%</td>
</tr>
<tr>
<td>Review</td>
<td>12%</td>
</tr>
<tr>
<td>Updating</td>
<td>11%</td>
</tr>
</tbody>
</table>

4.0 Summary

The Class Diagram (essentially an Entity Relationship Diagram) and the Event List or, better, the Statechart (a State Transition Diagram) seem to suffice to determine most of the development effort. The only activity we think is not covered is the development of complex operations (algorithms, methods). We suggest an approach based on COCOMO II to help the estimator identify complex operations (algorithms, methods) and estimate the additional effort. We have attempted to estimate the size in Unadjusted Feature Points.

There are still many questions to be answered. How do we account for the reuse of objects? How does productivity depend on the choices of architecture, physical design, and the development process? How accurate are the estimates? Should the estimates be inflated to cover objects not yet discovered? Future studies must address these and other questions.
References


Possible UML-Based Size Measures