Dynamic Metrics for Object Oriented Designs[1]

Sherif M. Yacoub, Hany H. Ammar, and Tom Robinson
Department of Computer Science and Electrical Engineering,
West Virginia University
Morgantown, WV26506
yacoub@csee.wvu.edu, hammar@wvu.edu, TRobinson@lcsys.com

Abstract

As object oriented analysis and design techniques become widely used, the demand on assessing the quality of object-oriented designs substantially increases. Recently, there has been much research effort to develop and empirically validate metrics for OO design quality. Complexity, coupling, and cohesion have received a considerable interest in the field. Despite the rich body of research and practice in developing design quality metrics, there has been less emphasis on dynamic metrics for object-oriented designs. The complex dynamic behavior of many real-time applications motivates a shift in interest from traditional static metrics to dynamic metrics.

This paper addresses the problem of measuring the quality of object-oriented designs using dynamic metrics. We present a metrics suite to measure the quality of designs at an early development phase. The suite consists of metrics for dynamic complexity and object coupling based on execution scenarios. The proposed measures are obtained from executable design models. We apply the dynamic metrics to assess the quality of a pacemaker application. Results from the case study are used to compare static metrics to the proposed dynamic metrics and hence identify the need for empirical studies to explore the dependency of design quality on each.

Keywords: Dynamic Metrics, Design Quality, Object-Oriented Designs, and Real-Time OO Modeling

1. Introduction

Object-oriented technology is gaining substantial interest as a beneficial paradigm for developing software applications. To evaluate the quality of OO software, we need to assess and analyze its design and implementation using appropriate metrics and evaluation techniques. As a result, several quality metrics have been introduced to measure the application quality at various development phases. Assessing the application quality at an early development phase is beneficial in guiding the development effort at subsequent phases. In this paper, we are concerned with metrics that are applicable at the design phase.

Several approaches have evolved to model OO designs. The *Unified Modeling Language* [24] is the result of the unification process of earlier OO models and notations. UML models capture the application static and dynamic aspects, but do not provide dynamic model execution or simulation. The *Real-Time Object Oriented Modeling* (ROOM) [23] was introduced to study the dynamic aspects of applications constructed as concurrently executing objects. ROOM design models support simulation of the application execution scenarios.

The metrics proposed in this paper are based on executable OO design models from which the dynamic behavior of applications can be inferred. The importance of evaluating the run time structure at the design level is recently pointed out by Gamma *et.al.* [10, pp23] "*The system's run-time structure must be imposed more by the designer than the language. The relationship between objects must be designed with great care, because they determine how good or bad the run-time structure is*". The excess effort in developing executable designs is justifiable for many complex real-time applications. These applications are usually modeled as executable designs prior to deployment in a working environment. Executable models are used to simulate real time applications and deduce their real-time properties such as deadlines and scheduling. The same models can be used to analyze the quality and criticality of objects prior to detailed implementation.

A quality metric should relate to external quality attribute of a design [14]. External attributes include maintainability, reusability, error-proneness, and understandability. Based on observations and empirical studies, coupling [3] and complexity [21] were shown to have a direct impact on software quality. As a result, we focus on coupling and complexity metrics. We identify a set of new dynamic metrics and discuss their relationship quality attributes.

1.1 Static versus Dynamic

Several metrics have been proposed to measure object oriented design quality. Design metrics can be classified into two categories; *Static* and *Dynamic*. To distinguish *static* and *dynamic* metrics, consider the two cases discussed by Hitz *et.al.*[14]; 1) class c_1 invokes a method

[1] This work was funded by the DoD research grant No. DAAH04-96-1-0419, monitored by the Army Research Office, to West Virginia University.

m of class c_2 10 times, and 2) class c_1 invokes 10 methods in 10 different classes, once each.

The first case measures the "*number of method invocations*" while the second measures the "*number of methods invoked*". Briand *et.al.* [4] identified the second case as a common metric and using the first metric would "*distort the value of the measure*". In this paper, we distinguish the first case as a dynamic metric of designs that can be used to identify critical components.

The dynamic aspects of design quality are less frequently discussed in OO quality metrics literature as compared to static metrics. The distinction between class level coupling (CLC) and object level coupling (OLC) was first made in [13]. The Coupling Between Object classes (*CBO*) metric [5] cannot be considered an object level coupling because "*for object level coupling it is not sufficient to know how many invocations from one method to another, but how often these invocations will be executed at run time*"[4]. *CBO* was defined as coupling between object *classes* and cannot be considered coupling between instantiated objects.

Currently, to the best of our knowledge, there is no literature available on object level coupling and dynamic coupling metrics. Briand *et.al.*[4] surveyed current literature on coupling metrics and concluded that "*All measures are defined to measure class-level coupling. No measure of object level coupling has been proposed.*" The authors further attributed this result to the difficulty in measuring coupling between individual objects. "*One way to measure coupling between objects would be to instrument the source code to log all occurrences of object instantiations, deletion, method invocations, and direct reference to attributes while the system is executing However, this kind of measurement can only be performed very late in the development process.*" In this paper, we define dynamic metrics that are obtained at an early development phase from executable design models such as ROOM models prior to detailed implementation phases.

1.2 Motivation

- Distinguish static and dynamic metrics by differentiating between measuring what is actually happening (dynamic) rather than what may happen (static).
- Assess the run-time quality of object oriented applications at the design level using executable design models instead of implementation phase assessment using source code.
- Active objects are sources of errors because they execute more frequent and experience numerous state changes that translate to attribute modifications and method invocations. Hence, we are motivated to assess the complexity of objects as expected at run-time.

1.3 Contributions

A recent study on coupling metrics surveyed coupling literature in OO systems and concluded that no measures for object level coupling are currently available [4]. This was attributed to difficulties in measuring coupling between individual objects. We perceive that given advances in object oriented modeling and simulation tools, we are able to evaluate object coupling and complexity from executable designs, which gives early measures of the design quality. In this paper, we define a set of dynamic metrics for object coupling and dynamic complexity, then we describe how these metrics affect the quality of an application. A case study is used to show why dynamic metrics are required in addition to existing static metrics. Consequently, the need for empirical studies, to evaluate the dependency of design quality on dynamic metrics, is inevitable.

This paper is organized as follows. First, we distinguish related work that could possibly appeal to be considered dynamic metrics and show how they actually reflect static measures. Section 3 summarizes the terminology and definitions used to express the proposed metrics. In Section 4 and 5, we discuss the proposed coupling and complexity metrics respectively. In Section 6, we use a case study of a *pacemaker* application to compare static and dynamic metrics. Finally, we conclude the paper and discuss possible future work.

2. Related Work

The problem of assessing the design quality of OO systems has been of interest to many researchers [5, 13, 14, 15, 2, and 4]. Coupling, cohesion, and complexity are defined as quality measures for OO systems. Chidamber and Kemerer [5] defined coupling between classes in object oriented applications. The relevance of coupling as a metric of design quality was then related to maintenance, testing, and understandability [13]. The suite represented in [2] classifies the coupling metrics according to the locus of impact, the relationship, and the interaction between classes. Empirical validation of these coupling measures, as related to error-prone software, was conducted by Briand *et. al.* [2, 3].

Many of the coupling measures defined in the literature are accurately estimated at the code level, explanation of the importance of assessing these quality measures at the high design level is discussed in [3]. Fenton [8, 9] discussed the role played by the measurement theory to scientifically describe a software measurement process. Based on this approach, an evaluation of Chidamber and Kemerer metrics was presented by Hitz and Montazeri [15], who also showed some useful guidelines in evaluating a measurement, and

described some steps to ensure that the measurement achieves its objective.

2.1 Coupling Metrics

This section discusses metrics that could mistakenly be considered as dynamic metrics. As a result of this discussion, it becomes more obvious why current coupling metrics cannot be considered dynamic metrics.

Briand *et.al.*[2] defined the *Method-Method* interaction (*MM*) for two classes *c* and *d* when "*a method implemented at class c statically invokes a method of class d (newly defined or overwritten), or receives a pointer to such a method*." From the definition it is clear that this is calculated from the class methods or signatures, which doesn't count the frequency of invocations.

Lee *et.al.*[17] defined the *Information Flow-based Coupling* (*ICP*) as the counts of the number of invoked methods of other classes weighted by the number of parameter. It is a sort of measure using information flow between classes but it is also calculated from the class methods not from object invocations.

Li *et.al.* [18] defined the *Message Passing Coupling* (*MPC*) as the count of the number of send statements that is found in methods of one class to other classes. Counting the number of *send* statements does not reflect the actual number (frequency) of execution of that send statement.

Chidamber *et.al.* [5] introduced the *Response for Class* (*RFC*) as a measure of the number of methods that can potentially be executed in response to a message received by an object of that class. Characterized as "*potentially*", this number does not reflect the actual invocations due to messages received by an object and hence is not a dynamic measure.

Chidamber *et.al.* [5] defined *Coupling between Object classes* (*CBO*) as "*the count of the number of classes to which it is coupled*" and further elaborated in the definition as "*two classes are coupled when methods of one class use methods or instance variables defined by the other class*". Obviously, it is a measure of coupling between object classes not between objects themselves (as the acronym *CBO* may indicate). This measure is not a dynamic measure of coupling because it does not count the number of invocations during execution, but it counts the number of methods and variables invoked.

Harrison *et.al* [12] have attempted an evaluation of the correlation between coupling and system understandability. Based on data from five object-oriented systems, they evaluated the correlation between *Coupling Between Objects* (*CBO*) and *Number of class Associations* (*NAS*). As a result of this experiment, they indicated that there exists a strong relationship between *CBO* and *NAS* and that coupling is not tightly related to system understandability. However, these results could be attributed to the sample case studies in which the difference between *CBO* and *NAS* numerical values is not distinct as well as the very loose coupling relationship between classes as stated in their paper. Given that we recognize *CBO* as a static measure, this study is a comparison between two static metrics.

Eder *et.al.* [7] extended classical concepts of coupling and cohesion in procedural-oriented to object oriented applications. They defined *Interaction Coupling* between methods of classes using the content, common, external, control and stamp degrees of coupling. *Interaction Coupling* could appeal to be considered a dynamic metric. However, it is not since it is obtained from classes not run-time objects. Moreover, the authors did not specify how to quantify the measures. Instead, a degree of coupling was proposed in terms of how complex and how explicit the coupling is.

The distinction between class and object coupling was made by Hitz *et.al.* [13] as *Class Level Coupling* (*CLC*) and *Object Level Coupling* (*OLC*). Hitz defined CLC as the dependency between classes at the development life cycle and OLC as the dependency between object-structure at run-time. OLC differs from CLC in excluding, as a coupling factor, access to a server object that is part of the calling object, an inherited object, or a local variable in the calling object methods i.e. native objects. Their work defines a framework for coupling classification and factors affecting object coupling. However, it lacks a quantitative component and formulae for coupling metrics between objects. They considered levels and degrees of coupling (strong, weak etc.) on an ordinal scale. The ordinal scale is not computationally suitable for mathematical ratios and sums and thus their approach does not support mathematical quantitative techniques.

2.2 Complexity Metrics

Software complexity metrics have been practiced in by scientists and engineers for a long time. In 1976, McCabe introduced cyclomatic complexity measurement as an indicator for system quality in terms of testability and maintainability. Cyclomatic complexity is based on program graphs and is defined as:

$$VG = e - n + 1$$

where: *e*=number of edges, *n*=number of nodes

There is a correlation between the number of faults and the complexity of the system [1, 21]. Therefore, static complexity is used to assess the quality of a software.

In the object oriented context, Chidamber *et. al.* [5] proposed a set of quality measures for class complexity.

They proposed *Depth of Inheritance Tree* (*DIT*), *Number of Children* (*NOC*), and *Weighted Methods Complexity* (*WMC*). Li *et.al.* [19] proposed other metrics for complexity based on size measures such as number of methods and attributes. In this context, complexity is related to the required maintenance effort as a result of modifications as well as understandability issues. These metrics are either based on the static view of the design; i.e., its class diagrams, or the source code of the classes. The metrics are used to evaluate the complexity of the design structure and hence are static metrics.

Most of these complexity definitions deal with the program at rest. However, the level of exposure of a module is a function of its execution environment. Hence, dynamic complexity [16] evolved as a measure of complexity of the subset of code that is actually executed.

Dynamic complexity was discussed by Munson *et.al.* [21] for reliability assessment purposes. The authors emphasized that it is essential to not only consider complex modules but how frequently they are executed. They defined execution profiles for modules that reflect what percentage of time a module is executing, and hence derived functional complexity and operational complexity as dynamic complexity metrics.

Ammar *et.al.* [1] extended dynamic complexity definitions to incorporate concurrency complexity. They further used *Coloured Petri Nets* models to measure dynamic complexity of software systems using simulation reports.

Here, we extend dynamic complexity metrics to measure the quality of object oriented designs. Complexity metrics are measures calculated for a particular object (instance of a class). Our approach is based on dynamic execution of the object's statechart behavior. The complexity metrics proposed is based on reports from simulation of executable designs.

3. Definitions and Terminology

o_i : is an instance of a class (an object)
O : is the set of objects collaborating during the execution of a specific scenario
$|Z|$: The number of elements in set Z
$\{\}$: A representation of a set of elements

Definition: A Scenario "x". The dynamic behavior of an object-oriented application can be specified using a set of scenarios. A scenario *x*, from a set of scenarios *X*, is a sequence of interactions between objects stimulated by input data or events.

Definition: Probability of a Scenario "PS_x". Each execution scenario has a certain probability of execution depending on the nature of the application. The probability of a scenario is the execution frequency of that scenario with respect to all other scenarios and is denoted as "PS_x"

Definition: A Scenario Profile. A scenario profile for an application is the set of probabilities. Each element of the set is the probability of execution of a scenario; i.e., PS_x

Definition: A Message set $M_x(o_i,o_j)$. $M_x(o_i,o_j)$ is set of messages sent from object o_i to object o_j during the execution of scenario *x*. A message is defined as a request that one object makes to another to perform a service. For the rest of the paper, we refer to an interaction between two objects as a message. Object interaction can be a method call, invocation or a message sent across a link.

Definition: Total Messages in a Scenario MT_x. MT_x is the total number of messages exchanged between objects during the execution of scenario *x*.

4. Dynamic Coupling Metrics

In this section, we define dynamic coupling metrics within a scenario scope; i.e. measurements are calculated for parts of the design model that are activated during the execution of a specific scenario triggered by an input stimulus. Then, we extend them to have an application scope; i.e., for all scenarios. We discuss the proposed coupling metric using the following template:

Context: The context in which the metric is applicable; i.e. in what situations can we apply the metric.

Description: A textual informal description of the metric that explains the meaning of the metric without overwhelming analytical details.

Formula: An analytical description of the metric using the terminology defined in the previous Section. The formula is useful in automating the process of obtaining measurements using the proposed metric.

Impact: An informal discussion on the impact of the metric on one or more of the design quality attributes such as: Maintainability, Understandability, Reusability, Error Proneness, and Error Propagation. The discussion in the impact section is based on intuition rather than theoretical proofs.

4.1 Export Object Coupling; $EOC_x(o_i,o_j)$

Context. A number of objects are instantiated from design classes and collaborate in a specific execution scenario to perform the application functionality.

Description. $EOC_x(o_i,o_j)$, the export coupling for object o_i with respect to object o_j, is the percentage of the number of messages sent from o_i to o_j with respect to the total number of messages exchanged during the execution of the scenario *x*

Formula

$$EOC_x(o_i, o_j) = \frac{|\{M_x(o_i, o_j) \mid o_i, o_j \in O \wedge o_i \neq o_j\}|}{MT_x} \times 100$$

Impact. *EOC* is a measure of mutual coupling between two specific objects. Mutual coupling between objects is important for identifying possible sources for exporting errors, identifying tightly coupled objects, and testing interactions between objects. *EOC* can affect the following design quality attributes:

Maintainability: A class whose object has higher *EOC* to another specific object would be more critical to changes due to maintenance and are more likely to export these maintenance changes to that specific object.

Understandability: An object sending many messages to another specific object (higher *EOC* count) is harder to understand because its dynamic behavior tightly depend on that particular object.

Reusability: An object with higher *EOC* to another specific object is less reusable because they tightly depend on another object and frequently request services from that particular object. The two objects are more likely to be used together.

Error Propagation: An object with high *EOC* is a source for error propagation since errors are more likely to propagate from the faulty source to the destination object as a result of the frequent messages sent to the destination.

Note. $EOC_x(o_i,o_j)$ can be extended to measure the percentage of the total number of messages sent by the object o_i to all other objects in the design. We call this the *Object Request for Service* (*OQFS)* and is given by:

$$OQFS_x(o_i) = \frac{|\{\bigcup_{j=1}^{K} M_x(o_i, o_j) \mid o_i, o_j \in O \wedge o_i \neq o_j\}|}{MT_x} \times 100$$

$$= \sum_{j=1}^{K} EOC_x(o_i,o_j)$$

A higher *OQFS* value could indicate that the calling object is active and hence sends many messages to other objects. *OQFS* also indicates that the object has complex behavior because it frequently invokes other objects to perform services.

4.2 Import Object Coupling; $IOC_x(o_i,o_j)$

Context. A number of objects are instantiated from design classes and collaborate in a specific execution scenario to perform the application functionality.

Description. $IOC_x(o_i,o_j)$, the import coupling for object o_i with respect to object o_j, is the percentage of the number of messages received by object o_i and was sent by object o_j with respect to the total number of messages exchanged during the execution of the scenario *x*.

Formula

$$IOC_x(o_i, o_j) = \frac{|\{M_x(o_j, o_i) \mid o_i, o_j \in O, o_i \neq o_j\}|}{MT_x} \times 100$$

Impact. Similar to *EOC*, *IOC* is a measure of mutual object coupling between two specific objects. They differ in the direction of coupling, which is classified as *Import* and *Export* [4]. The distinction between the direction of coupling is important to distinguish which object has an impact on another, the direction of possible error propagation, and possible impacts of changes due to maintenance. *IOC* can affect the following design quality attributes:

Maintainability: A class whose object has higher *IOC* to another specific object is more likely to import changes due to maintenance in the class from which that particular object is instantiated.

Understandability: An object receiving many messages from another specific object (higher *IOC*) is harder to understand because it provides many services to that specific object. However, it can also provide simple services that are frequently required.

Reusability: An object with higher *IOC* to another specific object is more likely to be reused with the other object using its services.

Error Proneness: Faults in objects with higher *IOC* could easily manifest themselves into failures of the applications because services of these objects are frequently required by other objects.

Note. $IOC_x(o_i,o_j)$ can be extended to measure the percentage of total number of messages sent to the object o_i from all other objects in the application during the execution of a specific scenario *x*. We call this the *Object Response for Service (OPFS)* and is given by:

$$OPFS_x(o_i) = \frac{|\bigcup_{j=1}^{K} M_x(o_j, o_i) \mid o_i, o_j \in O \wedge o_i \neq o_j|}{MT_x} \times 100$$

$$= \sum_{j=1}^{K} IOC_x(o_i, o_j)$$

4.3 Incorporating Scenario Profiles

The metrics in the previous sections are defined for a specific execution scenario. We can extend the scope of

the metric to incorporate the probabilities of execution of scenarios (or so called *Scenario Profiles*). We apply the metric to objects for a given scenario, then average the measurements weighted by the probability of executing the scenario. The metrics definitions, in an application scope, are:

$$IOC(o_i, o_j) = \sum_{x=1}^{|X|} PS_x \times IOC_x(o_i, o_j)$$

$$OPFS(o_i) = \sum_{x=1}^{|X|} PS_x \times OPFS_x(o_i)$$

$$EOC(o_i, o_j) = \sum_{x=1}^{|X|} PS_x \times EOC_x(o_i, o_j)$$

$$OQFS(o_i) = \sum_{x=1}^{|X|} PS_x \times OQFS_x(o_i)$$

where, X is the set of scenarios.

We can also use the metric to identify critical objects per scenario without averaging. In this case, objects are compared using measurements of a specific scenario. Critical objects for a specific scenario are also considered critical for the whole application.

4.4 Properties

Given the above definitions, we can deduce some properties and relationships between the proposed metrics as follows:

1) *EOC* and *IOC* are greater that or equal *zero*

2) $EOC_x(o_1,o_2) = IOC_x(o_2,o_1)$, the export coupling of one object with respect to another is the import coupling of the second object with respect to the first.

These properties explore some of the dependencies between the metrics. Principal Component Analysis (*PCA*) and empirical studies are required to resolve these dependencies.

4.5 Assumptions

These metrics are applicable to object oriented designs with executable models, prototypes, or could be simulated because they depend on the dynamic behavior aspects of the model. Existing modeling environments such as *Real Time Object Modeling* (ROOM) [23] makes this a valid assumption.

The mechanism by which a message is sent from one object to another is not of concern in defining the metric. Messages could be sent by direct method invocations, calls, or through specific links between objects. Access to an object attributes and methods are both treated as messages, we do not distinguish the two cases. We assume a fixed number of objects during execution of a scenario, object's destruction and creation are not considered.

5. Dynamic Complexity Metrics

Dynamic complexity metrics guide the process of identifying complex objects and hence trace these objects to classes from which they are instantiated. As a result, classes could be ranked based on the complexity of their instantiated objects. Implementation, testing, verification and validation efforts would be more dedicated towards high-complex classes than others.

The complexity model, we use here, uses the statechart behavioral specification of an object. Statecharts were first introduced by Harel [11]. Later, they were used for UML behavioral specification [6, 24] and were extended for real-time object modeling and simulation, ROOMcharts [23]. We base our dynamic complexity metrics on ROOMchart specification and use the simulation reports to calculate dynamic measurements.

The proposed dynamic complexity metrics extend our previous work in [1] for operational complexity [21] of modules based on *Petri Net* simulation reports. Similarly, in object oriented applications, we obtain dynamic complexity of objects based on ROOMchart specification and simulation results.

The operational complexity of objects is based on the static McCabe's cyclomatic complexity (*VG*) obtained from a control flow graph [20]. McCabe developed this non-primitive metric which measures the number of independent paths in a strongly connected graph. In such graph, the nodes represent the entry points, exit points, code segments, or decisions and edges represent control flow. First, we show how to construct flow graphs for ROOMchart specification of an object then we show how to obtain the operational complexity for that object.

5.1 Primitive Transitions

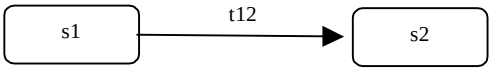


Figure 1 Simple Transition in ROOMchart

Figure 1 shows a simple transition *t12*. A transition in ROOMcharts specification is defined as:

transition *TransitionName*
triggered by *c1* **or** *c2*
action : { *action code* }

Thus, a transition is modeled by one or more conditions (also called triggers or guards) and by the action code segment that is executed when the transition

is fired. In ROOMcharts, each state has a code segment that is executed when entering and exiting the state. A state entry action is an action that is optionally associated with the state and is executed whenever this state is entered. A state exit action is an action that is executed whenever a transition is taken out of the state. When a transition is fired, the sequence of execution is; exit(s1), transition(t12), then entry(s2).

Thus, a transition is a flow of control that is guarded by the triggering condition. During this flow of control the entry, exit and action code segments are executed. Hence a control flow graph for the primitive transition of Figure 1 is shown in Figure 2.

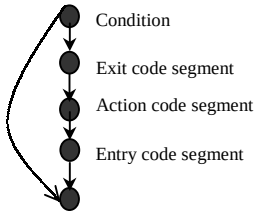


Figure 2 Control Flow for a Primitive Transition

The exit, action and entry code segments may be empty or it may contain detailed code. Thus, the complexity of these segments must be considered when obtaining the complexity of the flow graph of a transition. The cyclomatic complexity of a primitive transition of figure 2 is *VG =1*. But, the control flow graph of the entry, exit, and action code segments should also be considered. Thus, the cyclomatic complexity of the overall graph VG_t will be:

$$VG_t = VG + VG_x + VG_a + VG_e$$

where,

VG is primitive transition complexity,
VG_x is the complexity of the exit code segment,
VG_a is the complexity of the action code segment,
VG_e is the complexity of the entry code segment.

Figure 3 shows an example of the cyclomatic complexity of a transition whose exit and action code segments are significant while its entry code segment complexity is zero.

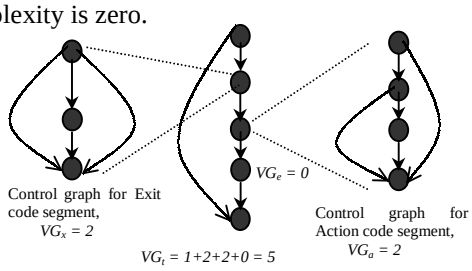


Figure 3 An Example

5.2 Initial Transitions

In ROOMcharts, when an object is created it takes an initial transition. The initial transition is the start of an execution thread that distinguishes active and passive objects. Like primitive transitions, an initial transition can have an action associated with it, thus its graph has an action code segment. The source of an initial transition is called *initial point*, which is not a state, thus its control flow graph does not have an exit code segment. An initial transition causes the object to enter a state. Hence its flow graph contains an entry code segment. Figure 4 shows an initial transition and its control flow graph, $VG_t = VG_a + VG_e$

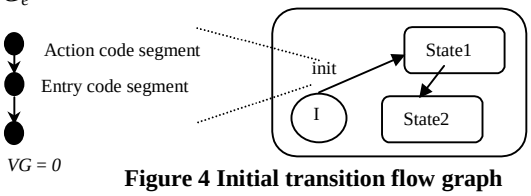


Figure 4 Initial transition flow graph

5.3 Transitions between Composite States

The large number of states in a state machine of a complex object can be reduced by chunking these states into separate sub-machines and packaging them within composite states. Composite states abstract lower level detailed behavior. A composite state is composed of lower level states. In ROOMcharts, transitions between composite states are cut into transition segments. A transition segment is part of the transition that belong to a composite state. Transition points provide a basis for correlating different segments of the same transition. To estimate VG_t for transition between composite states we sum the cyclomatic complexity of all transition segments, the exit code segment of the source node, and entry code segment of the destination node.

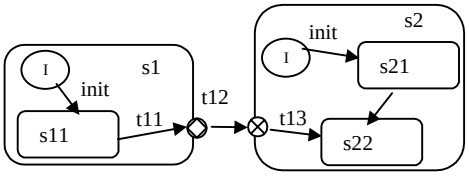


Figure 5 An example of a transition between composite states

For example, Figure 5, VG_t for the transition from state *s11* to *s22* is $VG_x(s11) + VG_a(t11) + VG_x(s1) + VG_a(t12) + VG_e(s1) + VG_a(t13) + VG_e(s22)$

5.4 Operational Complexity ($ocpx_x(o_i)$ and $OPCX(o_i)$)

For each scenario *x*, a subset of the ROOMchart model of an object o_i is executed in terms of state entries, state exits, and fired transitions. This subset of the ROOMchart model is translated into control flow graph as discussed in the previous subsections. Hence, we are able to calculate the cyclomatic complexity of the executed path for each object o_i for a scenario *x* ;i.e., $ocpx_x(o_i)$

Using the probabilities of execution scenarios (or operation profile), we obtain a measure of the operational complexity of the object from the simulation report. Using these reports, we identify the complexity of each object o_i for each scenario *x* and use the probabilities of scenarios to calculate the operational complexity of the object as follows:

$$OCPX(o_i) = \sum_{x=1}^{|X|} PS_x \times ocpx_x(o_i)$$

6. Case Study

We have selected a case study of a pacemaker device [6, pp177] to discuss the applicability of the proposed dynamic metrics and to illustrate a case where static quality metrics give little indication on the quality of the design. The pacemaker is a critical real-time application. An error in the software operation of the device can cause loss of the patient's life. Therefore, it is necessary to model its design in an executable form to validate the timing and deadline constraints. These executable models are also used, based on the proposed dynamic metrics, to categorize the criticality of objects based on their frequent execution, dynamic coupling, and dynamic complexity. We have used ObjecTime simulation environment [22] and ROOM models [23] to model and gather simulation statistics.

6.1 System Description

A cardiac pacemaker is an implanted device that assists cardiac functions when the underlying pathologies make the intrinsic heartbeats low. The pacemaker runs in either a programming mode or in one of operational modes. During programming, the programmer specifies the type of the operation mode in which the device will work. The operation mode depends on whether the Atrium (A), Ventricle (V), or both are being monitored or paced. The programmer also specifies whether the pacing is inhibit(I), triggered(T), or dual(D). For the purpose of this paper, we limit our discussion to the AVI operation mode. In this mode, the Atrial portion (A) of the heart is paced (shocked), the Ventricular portion (V) of the heart is sensed (monitored), and the Atrial is only paced when a Ventricular sense does not occur; i.e., inhibited (I). Figure 6 shows the pacemaker design model using a ROOM actor diagram. The pacemaker consists of the following actors:

Reed_Switch (RS): A magnetically activated switch that must be closed before programming the device. The switch is used to avoid accidental programming by electric noise.

Coil_Driver (CD): Receives/sends pulses from/to the device programmer. These pulses are counted and then interpreted as a bit of value zero or one. These bits are then grouped into bytes and sent to the communication gnome. Positive and negative acknowledgments as well as programming bits are sent back to the programmer to confirm whether the device has been correctly programmed and the commands are validated.

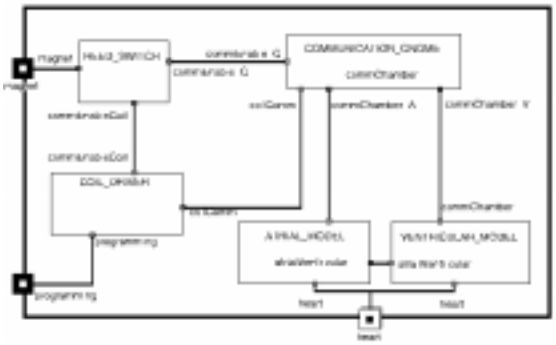


Figure 6 Actor diagram for the pacemaker example

Communication_Gnome (CG): Receives bytes from the coil driver, verifies these bytes as commands, and sends the commands to the Ventricular and Atrial models. It sends the positive and negative acknowledgments to the coil driver to verify command processing.

Ventricular_Model (VT) and *Atrial_Model (AR)*: These two actors are similar in operation. They both could pace the heart and/or sense heartbeats. The AVI mode, chosen to be simulated, is a complicated mode as it requires coordination between the Atrial and Ventricular models. Once the pacemaker is programmed the magnet is removed from the Reed_Switch. The Atrial_Model and Ventricular_Model communicate together without further intervention. Only battery decay or some medical maintenance reasons force reprogramming.

The behavior of each of the actors is modeled by a ROOMchart, which is based on statechart notation and semantics [11]. Due to space limitations, we have included selected ROOMcharts for some actors in appendix A.

As mentioned earlier, a pacemaker can be programmed to operate in one of several modes depending on which part of the heart is to be sensed and which part is to be paced. The analysis of the device operation defines

several scenarios. We limit our discussion to two scenarios. The first is *Programming* scenario in which the programmer sets the operation mode of the device. The programmer applies a magnet to enable communication with the device, then he sends pulses to the device which in turn interprets these pulses into programming bits. The device then send back the data to acknowledge valid/invalid program.

The second scenario is the *AVI_Operation* scenario. In this scenario, the Ventricular_Model monitors the heart. When a heart beat is not sensed, the Artial_Model paces the heart and a refractory period is then in effect.

The sequence diagrams for the two scenario are shown in appendix B. The programming scenario is executed less frequently than any operation scenario because the device is only programmed during maintenance periods which could be several months. Operation scenarios are triggered every heartbeat period of time.

6.2 Dynamic Coupling Metrics

Based on the two metrics *IOC* and *EOC* defined in Section 4, we use the simulation reports to gather the statistics for each actor in the system. We use the coupling matrix-approach [25] to explore dynamic coupling between the objects. The cell value of the coupling matrix indicates the import coupling measurement of the actor in the cell's row with respect to the actor in the corresponding column.

Using simulation reports, we develop Table 1 which shows the dynamic coupling matrix for the *Programming* scenario as percentages of the total number of messages sent during the scenario simulation.

	RS	CD	CG	AR	VT	Heart	Programmer
RS		0	0				8.3
CD	8.3		12.5				29.3
CG	8.3	16.7		0	0		
AR			8.3		0	0	
VT			8.3	0		0	

Table 1 Dynamic coupling for the *Programming* scenario in percentages (%)

Using simulation reports, we develop Table 2 which shows the dynamic coupling matrix for the *AVI_Operation* scenario as percentage of the total number of messages sent during the scenario simulation.

	RS	CD	CG	AR	VT	Heart	Programmer
RS		0	0				0
CD	0		0				0
CG	0	0		0	0		
AR			0		37.9	0	
VT			0	22.4		28.5	
Heart				11.2			

Table 2 Dynamic coupling for the *AVI_Operation* scenario as percentages (%)

Using Tables 1 and 2, we calculate *Object Request for Service* (*OQFS*) and *Object Response for Service (OPFS)* for each actor in the two scenarios. We then use scenario probabilities and the equations of Section 4.3 to produce Table 3 which shows the *OQFS* and *OPFS* in the system based on (0.01,0.99) scenario probabilities for *Programming* and *AVI_Operation* respectively.

	RS	CD	CG	AR	VT
OPFS	0.08	0.5	0.25	42.35	56.82
OQFS	0.27	0.27	0.46	53.53	45.47

Table 3 Dynamic Coupling for the pacemaker

6.3 Dynamic Complexity Metrics

Using the dynamic complexity metrics defined in Section 5, we gather statistics from the simulation of the two selected scenarios. Table 4 shows the dynamic complexity (as percentages) for the *Programming* scenario, the *AVI_Operation* scenario, and the dynamic complexity of the system based on (0.01/0.99) scenario probabilities.

Cpx for Scenario	RS	CD	CG	AR	VT
Programming	6.5	65.5	28	0	0
AVI	0	0	0	42.8	57.2
Overall	**0.065**	**0.655**	**0.28**	**42.4**	**56.6**

Table 4 Dynamic complexity for the pacemaker

6.4 Static Coupling Metrics

The following table shows the *Coupling Between Object classes* (*CBO*) [5] and *Number of Associations* (*NAS*) [12] for the pacemaker based on the actor diagram shown in Figure 6 and the number of actor invocation statements in the ROOM models. For the sake of comparison, the table shows the *NAS* for each actor as a percentage of the total number of associations. Similarly, the *CBO* is shown as percentage of the total *CBO* measurements for all actors.

	RS	CD	CG	AR	VT
NAS	18.75	18.75	25	18.75	18.75
CBO	13.5	23.1	28.8	17.3	17.3

Table 5 *NAS* and *CBO* for the pacemaker

6.5 Results

- From first row of table 5, we conclude that the coupling metric *NAS* cannot give a distinct indication of coupling levels in the system because the actors have *NAS* coupling levels that are close to each other except for the Communication_Gnome which is little higher.
- From second row of table 5, the coupling metric *CBO* gives an indication that the Communication_Gnome and the Coil_Driver are the most coupled actors followed by the Atrial_Model and Ventricular_Model.

This conclusion is based on a static metric, which does not consider the dynamic activities and frequency of messages between actors in the system.

- From table 3, the dynamic coupling metrics *OPFS* and *ORFS* indicate that the Atrial_Model and Ventricular_Model are the most coupled actors and coupling of other actors is not significant. This is because these two actors are the most active as they control the operational processing of the system. Hence, dynamic coupling metrics identified actors with the highest run-time coupling.
- From table 4, we conclude that the dynamic complexity of the actors Atrial_Model and Ventricular_Model are significantly higher than those for other actors. This is due to the fact that those two actors are active and executing most of the simulation time.

Static metrics, as applied to this example, either do not show significant ranking for coupling levels (the *NAS* metric), or has identified coupling level based on static non-executing models (the *CBO* metric). On the other hand, results from applying the proposed dynamic metrics have identified that the two objects Atrial_Model and Ventricular_Model are the most run-time coupled objects. As of dynamic complexity, the same two models are identified to be the most dynamically complex objects

As a result, we conclude that static and dynamic metrics can give different indications on the coupling and complexity of design elements. Dynamic metrics are concerned with frequently invoked and frequently executing objects while static metrics are concerned with statically coupled and complex design elements. Both metrics can give different indication on the quality of the design and empirical studies are essential to correlate each to errors found when the application is executed.

7. Conclusion

This paper augments existing literature on assessing the quality of object oriented designs by proposing a set of dynamic metrics as important supplements to existing static metrics. We define two metrics for object coupling (*Import Object Coupling* and *Export Object Coupling*) and a dynamic complexity metric (*operational complexity based on ROOMcharts*). Measurements using the proposed metrics can be obtained at early development phases from executable design models. The metrics are applied to a case study and measurements are used to compare static and dynamic metrics. We envisage that dynamic metrics are as important as static metrics for evaluating design quality and identify the need for empirical studies to assess the correlation between dynamic metrics and faulty modules.

From this work we conclude the following:

- Dynamic metrics can be used to measure the actual run-time properties of an application as compared to the expected properties measured by static metrics.
- To use the proposed dynamic metrics early in the development phase, the application has to be modeled as executable designs. The excess effort in developing executable designs is justifiable for many complex real-time applications where deadlines should be investigated prior to developing the application.
- The proposed metrics do not reflect the complexity or coupling to abstract classes. It is applicable to concrete classes from which objects are executing at run-time.
- Two phases are important for establishing a metric for dynamic quality. The first is defining metrics, differentiating them from exiting metrics, and identifying the need for these metrics. The second phase is using empirical studies to validate the metric. This paper addresses the first phase and identifies the need for the empirical studies to explore the correlation between dynamic metrics and design quality in one hand and dynamic and static metrics on another hand.

Future Work

- Explore the dependency between static and dynamic metrics. Objects are executable versions of the design, classes are the maintainable version. A correlation can exist between dynamic and static metrics.
- Empirically validate the proposed metrics and their correlation with design quality attributes.
- Develop a methodology for risk assessment based on dynamic metrics. Dynamic complexity metrics and severity measures will be used to categorize objects of high risk. Dynamic coupling will be used to assess how errors in objects affect other objects in higher risk categories.

References

[1] Ammar, H. H., T. Nikzadeh, and J. Dugan, "A Methodology for Risk Assessment of Functional Specification of Software Systems Using Coherent Petri Nets", *Proc. Of the Fourth International Software Metrics Symposium, Metrics'97*, Albuquerque, New Mexico, Nov 5-7, 1997, pp108-117

[2] Briand, L., P. Devanbu, and W. Melo, "An Investigation into Coupling Measures for C++," *Proc. Of the 19th International Conference on Software Engineering*, ICSE'97, Boston, May 1997, pp 412-421

[3] Briand, L., S. Morasca, V. Basili, "Defining and Validating Measures for Object-based High-level Design ", *Fraunhofer International Software Engineering Research*, Germany 1998, ISERN-98-04

[4] Briand, L., J. Daly, and J. Wurst, "A Unified Framework for Coupling Measurement in Object-Oriented Systems", *IEEE*

Transaction on Software Engineering, Vol 25, No. 1, Jan/Feb 1999, pp91-121

[5] Chidamber, S. , and C. Kemerer, "A Metrics Suite for Object Oriented Design ", *IEEE Transactions on Software Engineering*, Vol 20, No 6, June 1994, pp476-493

[6] Douglass, B., "Real-Time UML : Developing Efficient Objects for Embedded Systems", Addison-Wesley, 1998

[7] Eder, J., G. Kappel, and M. Schrefl, "Coupling and Cohesion in Object-Oriented Systems," Technical Report, University of Klagenfurt, 1994, ftp://ftp.ifs.uni-linz.ac.at/pub/ publications/ 1993/0293.ps.gz

[8] Fenton, N., "Software Metrics: A Rigorous Approach", London, Chapman&Hall, 1991

[9] Fenton, N., "Software Measurement: A necessary Scientific Basis", *IEEE Transaction on Software Engineering* Vol 20, No 3, 1994, pp199-206

[10] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Object-Oriented Software", Addison-Wesley, 1995.

[11] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, July 1987, pp231-274

[12] Harrison, R., S. Counsell, and R. Nithi, "Coupling Metrics for Object Oriented Design", *Proc. Of the 5th Metrics Symposium, Bethesda, Maryland, Nov 20-21*, 1998, pp150-157

[13] Hitz, M., and B. Montazeri. "Measuring Coupling and Cohesion in Object-oriented Systems", in *Proc. International Symposium on Applied Corporate Computing*, Monterrey, Mexico, Oct. 1995.

[14] Hitz, M., and B. Montazeri "Measuring Product Attributes of Object-Oriented Systems", in *Proc. 5th European Software Engineering Conference (ESEC `95)*. Barcelona, Spain: Lecture Notes in Computer Science 989, Springer-Verlag 1995, pp124-136

[15] Hitz, M., and B. Montazeri; "Chidamber & Kemerer's Metrics Suite, A Measurement Theory Perspective", *IEEE Transactions on Software Engineering*, Vol 22, No. 4, April 1996, pp276-270

[16] Khoshgoftaar, T. M., J. C. Munson, and D.L. Lanning,"Dynamic System Complexity", *Proc. Of International Software Metrics Symposium, Metrics'93*, Baltimore MD., May 1993, pp129-140

[17] Lee, Y., B. Liang, and S. Wu, "Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow," *Proc. Of International Conference on Software Quality*, Maribor, Slovenia, 1995

[18] Li, W., and S. Henry, "Object Oriented Metrics that predict Maintainability" *Journal of Systems and Software*, Vol 23, No. 2, 1993, pp 111-122

[19] Li, W., and S. Henry, "Maintenance Metrics that for the Object Oriented Paradigm" *Proc. Of 1st International Software Metrics Symposium*, Los Alamitos, CA, 1993, pp52-60

[20] McCabe, T., "A Complexity Metrics", IEEE Trans on Software Engineering, Vol 2, No. 4, Dec 1976, pp308-320

[21] Munson, J., and T. Khoshgoftaar, "Software Metrics for Reliability Assessment", in *Handbook of Software Reliability Engineering*, Michael Lyu (edt.), McGraw-Hill, 1996, Chapter 12, pp 493-529

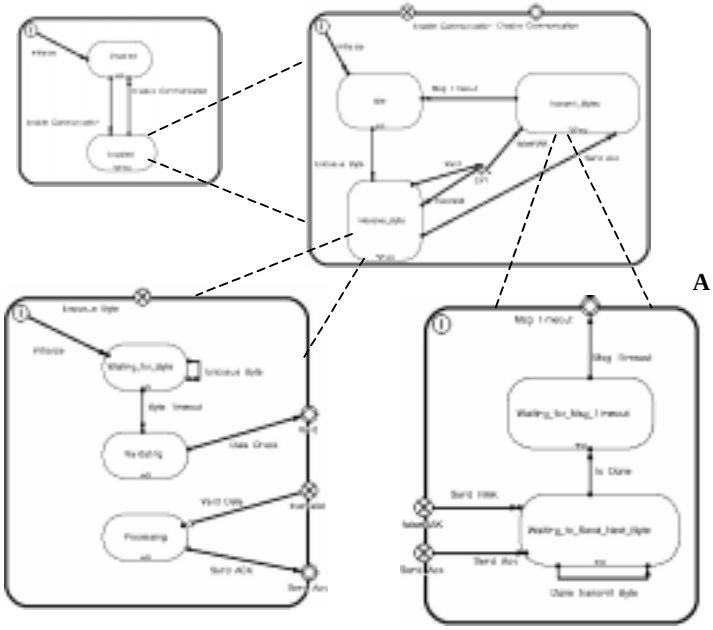
[22] ObjecTime User Guide. ObjecTime Ltd., Kanata, Ontario, Canada, 1998.

[23] Selic, B., G. Gullekson, and P. Ward, "Real-Time Object Oriented Modeling", John Wiley & Sons, Inc. 1994

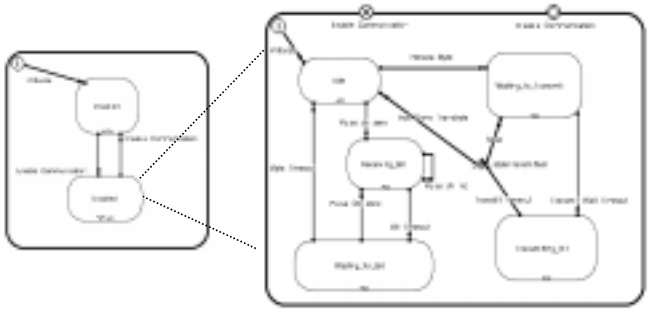
[24] The Unified Modeling Language Resource Center http://www.rational.com/uml/ index.html

[25] Yacoub, S., and H. Ammar, "A Matrix-Based Approach to Measure Coupling in Object-Oriented Designs", to appear in *Journal of Object Oriented Programming, JOOP*, 1999

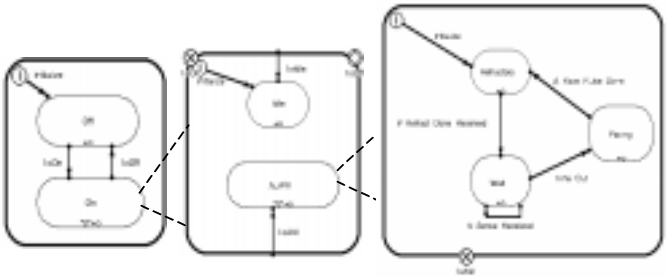
Appendix A : *ROOMcharts* for selected actors of the pacemaker example



A *ROOMchart* for Communication Gnome

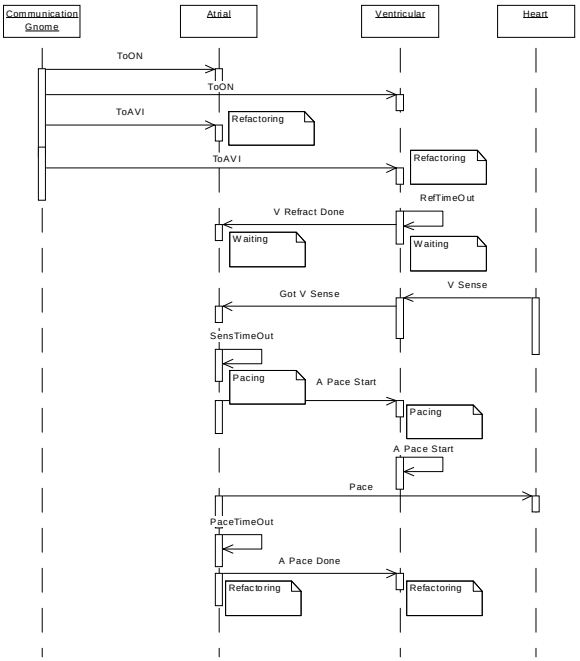


A *ROOMchart* for Coil Driver

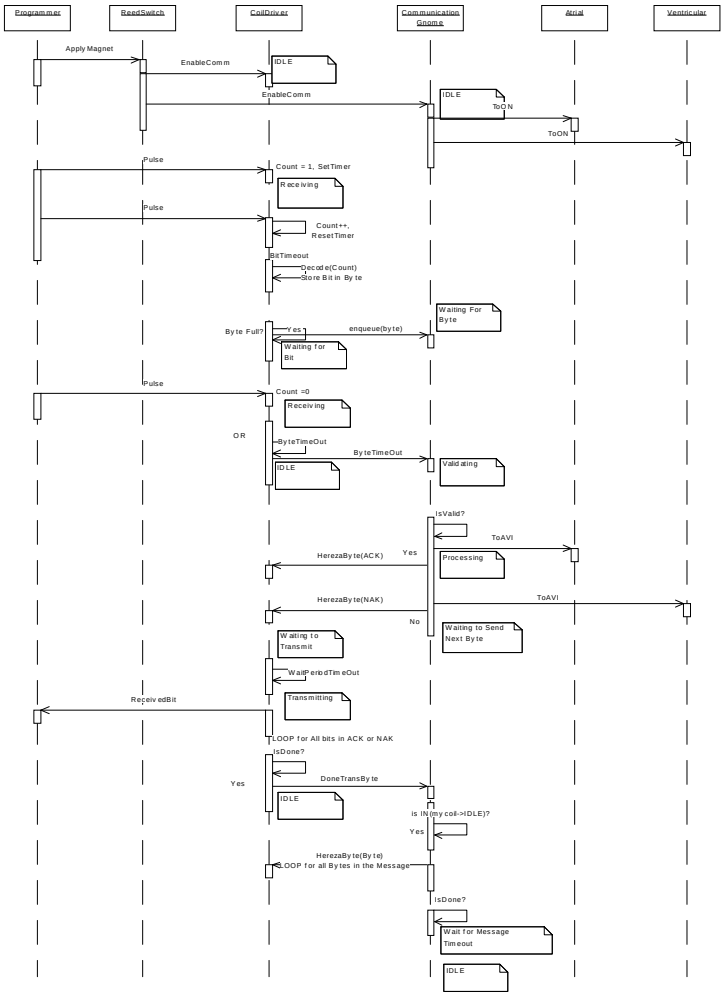


A *ROOMchart* for Atrial Model

Appendix B : Sequence Diagrams for selected Scenarios of the pacemaker example



Sequence Diagram for Programming Scenario



Sequence Diagram for AVI_Operation Scenario