Analytical Partition of Software Components
for Evolvable and Reliable MEMS Design Tools[*]

Carol L. Hoover
Department of Electrical & Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213-1890
clh@cs.cmu.edu

Pradeep K. Khosla
Institute for Complex Engineered Systems
Carnegie Mellon University
Pittsburgh, PA 15213-1890
pkk@ices.cmu.edu

Abstract

Transforming software requirements into a software design involves the iterative partition of a solution into software components. The process is human-intensive and does not guarantee that design objectives such as reusability, evolvability, and reliable performance are satisfied. The costly process of designing, building, and modifying high assurance systems motivates the need for precise methods and tools to generate designs whose corresponding implementations are reusable, evolvable, and reliable. This paper demonstrates an analytical approach for partitioning basic elements of a software solution into reusable and evolvable software components. First, we briefly overview the role of partitioning in current design methods and explain why computer-aided design (CAD) tools to automate the design of microelectromechanical systems (MEMS) are high assurance applications. Then we present our approach and apply it to the design of CAD software to layout an optimized design of a MEMS accelerometer to be used in the navigational units of aircraft. Lastly, we discuss the implications of our approach and future research directions.

1. Introduction

The process of designing the software for a high assurance system is a tedious and human-intensive process. Design is a transformation from software requirements into a specification of the modules to be built. Software design involves the repeated partition of a solution into components. The partition process starts with the identification of basic system and subsystem (high-level design) components and concludes with the identification of modules (low-level design components) to be implemented. For our discussion,

the term partition has two related but different definitions: (1) the process of dividing into parts, and (2) the process of grouping elements into disjoint sets. We specifically apply the first definition to the decomposition of a software solution and the second definition to the logical grouping of solution elements such as data and operations.

Prevailing design methods provide guidelines but minimal automated support for the systematic partition of a software solution into components that satisfy design constraints. Commercial tools support the documentation of software designs but provide only limited support for analyzing the "goodness" of a design. A good software design not only satisfies the required business objectives but also promotes design objectives such as reuse of software components over time and evolution of the resulting software system. In this paper, we analyze the design and evolution of a new class of high assurance systems: the computer-aided design (CAD) tools for designing microelectromechanical systems (MEMS) devices.

CAD software is a high assurance application because its failure to generate suitable MEMS designs would increase the cost of prototyping a MEMS device and could potentially impact the performance of the device. MEMS devices are used in automobile air bags and experimentally in aircraft navigational units. The construction of accelerometers and gyroscopes from MEMS devices could potentially yield a reduction in cost, weight, volume, and power consumption in comparison to those built using traditional laser and fiber optic technologies.

In the case of CAD tools to design MEMS, "goodness" refers to a CAD tool's capability to generate a blueprint for a device that performs accurately without modification. The high cost of prototyping MEMS devices mandates simulation of the device behavior [17]. During the design of a MEMS device, the human uses the CAD tool to iteratively layout and simulate different blueprints. The human may follow design rules such as those defined by the Multi-User MEMS Processes or MUMPs [15]. Current software tools help automate different parts of the MEMS design

process, but they are not fully automatic and are not well integrated. For instance, some research tools generate optimized designs for MEMS accelerometers with fixed shapes but require another tool to simulate the behavior of the device. Future CAD tools should be able to automatically generate and test blueprints for accelerometers whose shapes are determined by a randomized search of the design space. The question is how to design the CAD software to support an integrated process for designing MEMS devices and to enable the evolution of MEMS design technology.
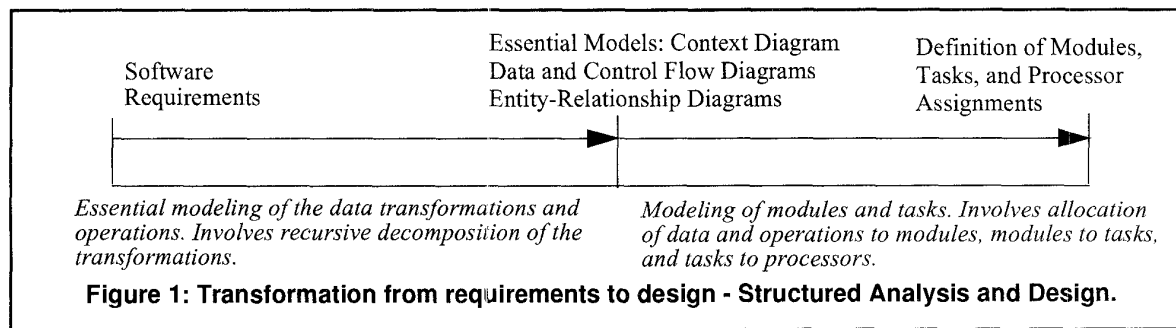
As introduced in a seminal paper [18], many design methods recommend that the designer isolate in separate modules those parts of the system which are to be reused or changed. For an example application, the reader should refer to a case study of the guidelines used by the U.S. Naval Research Laboratory (NRL) to modularize the A-7E avionics systems [2]. This study is based on the original NRL report by [6]. The problem is that the burden of determining how to best localize solution elements for reuse and evolution is on the designer: commercial tools for automatically partitioning solution elements do not exist.
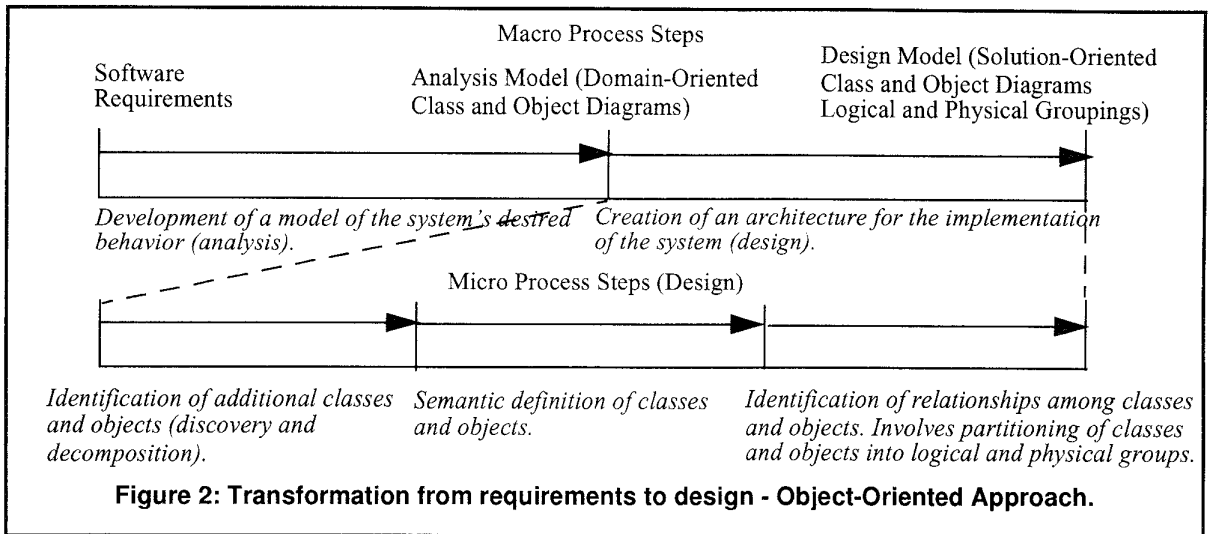
In this paper, we present an analytical method that we developed to partition basic elements of a software solution into reusable and evolvable design components that can be mapped to implementable modules. Our approach directs the designer to carefully denote elements for potential reuse or change and uses this information to mathematically determine a partition. In *Section 2*, we briefly overview the role of partitioning in two popular types of design methods: structured design and object-oriented design. *Section 3* is a brief review of research related to our work. In *Sections 4* and *5*, we discuss the application of our approach to the design of a software package for laying out and simulating MEMS accelerometers. *Section 6* includes a discussion of the results of applying our approach, and *Section 7* concludes with future research directions.
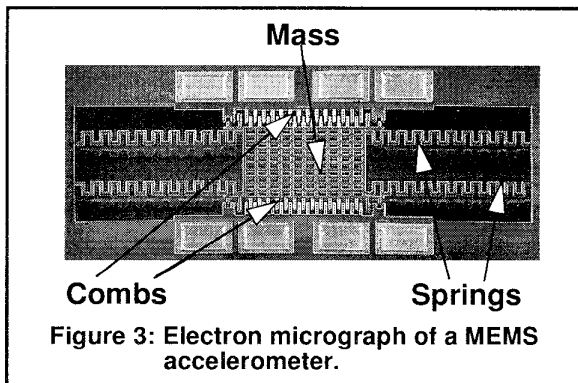
2. Background

In this section, we discuss the importance of partitioning (decomposition and grouping) in the design process. *Figures 1 and 2* show a macro-level view of the transformation from requirements to design for structured and object-oriented types of design. With both types of methods, the designer is responsible for determining the appropriate level of decomposition and the grouping of solution elements to best satisfy design objectives. The two specific methods that we reference, Ward and Mellor's structured development method and Booch's object-oriented analysis and design method, are widely used and representative of each type of design method [23], [5]. The terms decomposition, allocation, and grouping indicate the extensive and iterative role of partitioning in the transformation from requirements to design.

Software Requirements	Essential Models: Context Diagram Data and Control Flow Diagrams Entity-Relationship Diagrams	Definition of Modules, Tasks, and Processor Assignments
Essential modeling of the data transformations and operations. Involves recursive decomposition of the transformations.		*Modeling of modules and tasks. Involves allocation of data and operations to modules, modules to tasks, and tasks to processors.*

Figure 1: Transformation from requirements to design - Structured Analysis and Design.

189

	Macro Process Steps	
Software Requirements	Analysis Model (Domain-Oriented Class and Object Diagrams)	Design Model (Solution-Oriented Class and Object Diagrams Logical and Physical Groupings)

Development of a model of the system's desired behavior (analysis). *Creation of an architecture for the implementation of the system (design).*

Micro Process Steps (Design)

Identification of additional classes and objects (discovery and decomposition). *Semantic definition of classes and objects.* *Identification of relationships among classes and objects. Involves partitioning of classes and objects into logical and physical groups.*

Figure 2: Transformation from requirements to design - Object-Oriented Approach.

The identification of user-oriented objects (classes) and solution-oriented objects (classes) involves abstraction as well as partitioning. MEMS accelerometers consist of four basic structures: (1) a frame, (2) a mass, (3) a set of parallel metal teeth (combs) that outline part of the mass, and (4) springs attached to the mass and the frame of the device. *Figure 3* shows an electron micrograph of a MEMS accelerometer.

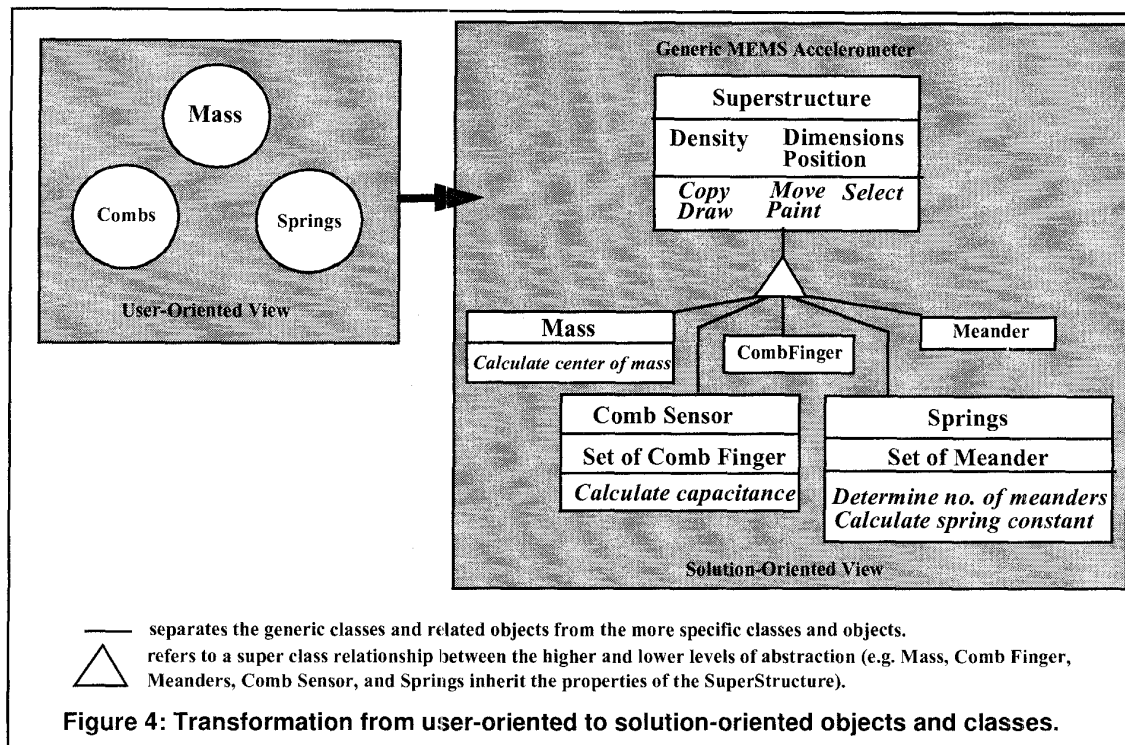**Figure 3: Electron micrograph of a MEMS accelerometer.**

Some background information about accelerometers follows. In an accelerometer, the springs allow the mass to move within the stationary frame. When the mass is subjected to an acceleration, it moves and causes the springs to expand or contract. The springs, in turn, exert a restoring force on the mass, where the spring constant k depends on the material of the springs and their shape [9]. The net result is that the distance moved by the mass is proportional to the external acceleration. The ensuing change in capacitance of combs is proportional to the distance moved and therefore proportional to the acceleration [16].

In *Figure 4* on the next page, we show the transformation from a user-oriented view of a generic MEMS accelerometer to the solution-oriented view of this device. Following the Booch method, the designer might decompose the user-oriented Mass, Combs, and Springs objects into a generic SuperStructure and more specialized objects which hide the details of the specific structures used in a MEMS accelerometer. The decision to decompose objects into generic super-objects (classes) and more detailed objects (classes) is solely dependent on the view-point of the designer. There are guidelines but no precise process for making such decisions.

For an in-depth report on an object-oriented analysis of another high assurance application, the air traffic control domain, the reader should see [24]. The report outlines the results of a research project which applied the Booch method to model the data for advanced flight plan processing systems. The Computer Information Systems group at the Technical University of Berlin and the EUROCONTROL Experimental Center (EEC) jointly performed the work.

190

Figure 4: Transformation from user-oriented to solution-oriented objects and classes.

In the figure:

——— separates the generic classes and related objects from the more specific classes and objects.

△ refers to a super class relationship between the higher and lower levels of abstraction (e.g. Mass, Comb Finger, Meanders, Comb Sensor, and Springs inherit the properties of the SuperStructure).

3. Related Research

The research and development of styles or patterns of architecture are efforts to codify the knowledge of the expert designer. Novice designers would use a handbook of patterns or styles to guide them in the selection of a type of design that has been successively used to solve a similar problem ([22], [10], and [7]). The description of the style or pattern includes a definition of the problem, the forces which guide or constrain the solution to the problem, and the solution. The solution or software design consists of components and their interactions. Some researchers are developing languages for describing patterns. For instance, the ROOM approach defines a model and language for documenting architectural patterns for real-time systems [21]. The styles or patterns approaches depend upon the expert designer to synthesize and codify good designs.

The SAAM, Software Architecture Analysis Method, is a technique for evaluating a candidate software architecture with respect to quality attributes such as modifiability and performance. The human evaluator measures an architectural description with respect to an agreed upon set of scenarios of how the system will be used. In effect, the candidate architecture is given a qualitative rating based on its perceived ability to support each scenario. Candidate architectures are then compared to each other with respect to

how they "perform" for similar scenarios [2]. Like the styles or patterns approach, SAAM requires the designer to synthesize the candidate architectures.

Our goal is to make the process of synthesizing a software architecture more systematic and automatable. We mathematically model the relationships between basic solution elements according to design objectives such as reuse, evolution, and reliability. Then we use these relationships to determine a good partition of the solution elements. Similarly the hardware-software codesign community partitions basic functional units for implementation in hardware or software and for allocation to different processing elements [1]. But unlike the codesign approach, we do not start with a set of predefined functional units. Instead, we transform a high-level definition of a software solution into design components that can be mapped to implementation modules. Our design space is more diverse and not so well defined.

4. Partitioning Data and Operations

The goal for our study is to apply our partitioning approach to the synthesis of an architecture for a MEMS design tool that could be easily upgraded as the MEMS technology evolves. The expectation is that CAD tools will become increasingly automatic as research in MEMS de-

191

sign provides mathematical models for relating the structural features of a MEMS design to the performance of the corresponding device. For this paper, we focus on the software tool support needed to design MEMS accelerometers. We show how this support will evolve and the impact that this evolution will have on the architecture of the design tool.

Basic elements of a software solution include data, operations, and control flow. Following a structured approach, the designer identifies the primary data elements (denoted as data stores) and processes that transform the data. In the object-oriented approach, the designer identifies objects, some or all of which encapsulate key data for the software solution. Encapsulating data and operations, or information hiding, supports the localization of solution elements that change together. The problem is that the designer is solely responsible for determining the composition of the basic system components or objects. Prevalently used design methods do not guarantee that the designer will consider the appropriate level of reuse or group together those elements which change together. Our approach has two primary features: (1) a manual but guided reuse and change analytic process and (2) a mathematical model and automatable algorithm for localizing solution features that change together. Shown below are the six basic process steps adapted from our original approach discussed in [13].

1. *Identify the basic data and operational features of the software solution.*
2. *Recursively decompose the large-effect operations and identify additional data elements.*
3. *Enumerate the feasible or expected changes to the software solution.*
4. *Determine the change set of data and operations for each expected change.*
5. *Combine and componentize the overlapping change sets.*
6. *Add other necessary components.*

Step 1: Identify the basic data and operations.

The first step in our approach is to identify the basic data and operational features of the software solution. The MEMS designer will interact with the tool to perform the following basic operations.

1. *Assign the structural parameters for the accelerometer (e.g. Young's modulus of material for the springs, the densities of the mass and the springs, and the workspace area).*
2. *Create an initial blueprint for the accelerometer.*
3. *Simulate the sensitivity of the accelerometer as determined by its ability to detect the change in capacitance of the combs as the mass accelerates.*

4. *Compare the calculated sensitivity resulting from the simulation to the required sensitivity.*
5. *Adjust the blueprint for the accelerometer.*

The basic data includes the structural parameters and the description of the blueprint. As discussed in *Section 2*, the essential elements of a MEMS device are the mass, combs, and springs. Therefore, the blueprint will also contain data representations for each of these structural elements.

Step 2: Decompose the large-effect operations and identify additional data elements.

The second step involves the decomposition of the large-effect operations into smaller-effect operations. The process repeats recursively until further decomposition results in operations which are trivial and therefore not reusable or which do not help to make the solution easier to understand and design. For instance, MEMS design operations 2 and 5, creating and adjusting the blueprint for an accelerometer can both be decomposed into the sub-operations of laying out the mass, laying out the combs, and laying out the springs. The process of laying out each of the essential structures will vary depending on the shape that the designer draws for the mass. The layout operations should be changeable for differently shaped masses and reusable across other blueprints requiring similarly shaped masses. The layout operations are reusable but are also complex and contain sub-operations which are reusable. Therefore, we decompose the layout operations to determine the reusable sub-operations. In the case of laying out the mass, we discover basic graphical operations and the operations to calculate the mid-point or the center of mass. The graphical operations inherent in all of the layout operations are the following.

- Positioning the cursor.
- Drawing a structural body.
- Painting a structural body.
- Selecting a structural body.
- Copying a structural body.
- Inverting a structural body.

The related data are the cursor and a generic structural body.

Here our approach goes beyond that used in our previous study [13]. Previously, we decomposed an operation T into a set of sub-operations if and only if all of the sub-operations of T are reusable. Now we decompose T if and only if at least one sub-operation t is reusable. We add T to the set of reusable operations if at least one sub-operation t of T is not reusable. The result is that we include larger-effect operations with definite potential for reuse along with the smaller-effect operations in the resulting set of operations.

Likewise in another example, decomposition of the MEMS design operation to simulate the sensitivity of the accelerometer yields the reusable sub-operation of inputting the frequency and amplitude of the sinusoidal wave and the number of cycles. The other sub-operations resulting from the decomposition, listed below, are not reusable.

- Determine the displacement per unit of time and use this value to calculate the change in capacitance of the combs.
- Calculate the signal-to-noise ratio and alter the input signal appropriately.

Therefore we include the sensitivity calculation and the input of the simulation parameters in the set of reusable operations, but we do not include the calculation of the change in capacitance of the combs or the calculation of the signal-to-noise ratio which will be encapsulated by the sensitivity calculation.The resulting sets of data and operations are represented mathematically in *Figure 5*.

$D = \{d \Diamond DataItem\}$

$\quad = \{mass, combs, springs, structuralBody, cursor, ...\}$

$O = \{o \Diamond operator\}$

$\quad = \{InputStructParms, LayoutMass, LayoutCombs, ...\}$

$x \Diamond xtype$ means that x is of type *xtype*.

Figure 5: Mathematical representation of steps 1 and 2 results.

Step 1 - Identify basic data and operations.
Step 2 - Decompose large-effect operations and identify additional data elements.

Step 3: Enumerate the feasible or expected changes to the software solution.

The third step is to enumerate the types of changes that the researcher would expect to make to the software solution. Prime candidates are changes to the data and operations that were identified in steps one and two. Determining expected or feasible changes to a software solution requires the designer to think critically about both the problem and the solution domains. Enumerating these changes is more than a prediction activity. The designer should talk with the domain expert or requirements analyst about changes in requirements that may be planned as well as those which are feasible though not specifically planned. The designer can then consider the changes to the solution that would be necessary to support the changes in requirements.

Changes that are likely to be made to the process of laying out a MEMS accelerometer include the following.

- Change the representation of the mass (e.g. change in shape).
- Change the representation of the springs.
- Change the representation of the combs.
- Construct a spring with more than one anchor point to the mass.
- Change from a table-driven to a calculated spring constant.
- Replace a calculation of the mid-point of a rectangular mass with a calculation of the center of mass for a non-rectangular mass.
- Change the layout of the mass.
- Change the layout of the combs.
- Change the layout of the springs.
- Modify the calculation of the change in capacitance of the combs.
- Replace the manual layouts of the mass, combs, and springs with an automatically generated layout of the whole MEMS device.
- Change mechanism for the input/output of the structural parameters.
- Change mechanism for the input/output of the parameters for simulating the sensitivity of the accelerometer.
- Change the graphics operations.

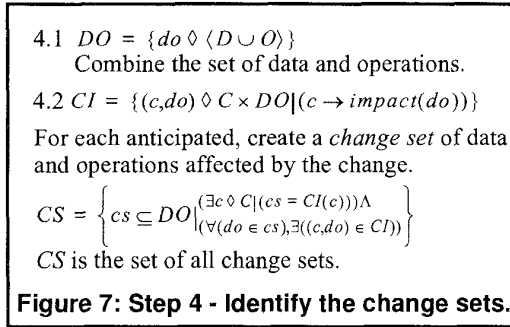We then represent these mathematically as a set C of expected changes as shown in *Figure 6*.

$C = \{c \Diamond change\}$

$\quad = \{ChgMassShape, ChgSpringsRep, ChgCombsRep, ...\}$

Figure 6: Step 3 - Enumerate the expected changes to the software solution and create a set of changes.

Step 4: Determine the change set of data and operations for each expected change.

The fourth step involves the partition of the data and small-effect operations into software components based on the analysis of anticipated changes to the software solution. The goal is to group data and operations affected by the same changes into the same components. The expected changes to the software design or its implementation can then be made simply by replacing whole components or by modifying a minimal number of components whose contents are affected by the changes. The idea is to mathematically relate the parts of the software solution that would actually have to be modified in order to accomplish a specific change. Those parts related to the same change either directly or transitively will be localized in the same component.

Step four consists of two substeps, 4.1 and 4.2, as outlined in *Figure 7*. First we combine the data and operations into a set *DO*. Then we relate each anticipated change to the software solution to the data and operations whose implementations would be affected by this change via a change impact relation CI. Applying the relation *CI*, we obtain a set of data and operations for each expected change. We will call these *change sets*. In totality, we now have a set of change sets *CS*.


4.1 $DO = \{do \lozenge \langle D \cup O \rangle\}$
   Combine the set of data and operations.

4.2 $CI = \{(c,do) \lozenge C \times DO | (c \rightarrow impact(do))\}$

For each anticipated, create a *change set* of data and operations affected by the change.

$$CS = \left\{ cs \subseteq DO \Big|^{(\exists c \lozenge C | (cs = CI(c))) \wedge}_{(\forall (do \in cs), \exists ((c,do) \in CI))} \right\}$$

CS is the set of all change sets.

Figure 7: Step 4 - Identify the change sets.

A data or operation is related via the impact relation *CI* to a proposed change in requirements if it will need to be modified to satisfy the change. The designer can start by considering how the data or operations might change. The table in *Appendix A: Impact of Feasible Changes* contains a list of the changes that are feasible to support the evolution of MEMS design technology. For each change *c* in the table, we list the set *cs* of data and operations whose design and corresponding implementation would have to be modified to support the corresponding change in requirements. Each change set *cs* is labeled with a *change signature* such as **CSM** which represents changing the shape of the mass.

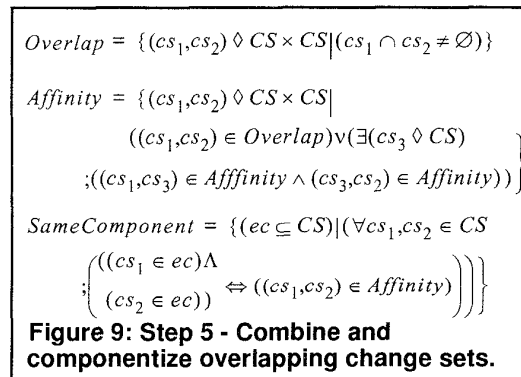**Step 5: Combine and componentize the overlapping change sets.**

We might think that we are done in that each set of data and operations affected by a change could be the contents of a software component. Then in the future, all we would have to do to make the expected change is to exchange or modify the related software component. But such compositions could result in duplicate copies of data or operations. In the above table, **CRS**, **CSAP**, and **CSC** impact the "calculate spring constant" operation. What is needed is a way to combine change sets that intersect one another. In the fifth step, we define a relation called *Overlap* that associates two change sets that have a non-empty intersection. The formal representation of this relation appears in *Figure 9*.

We must also consider the case when change sets are related transitively through repeated application of the *Overlap* relation. As shown in *Figure 8*, the change set **CRS** is

related to the change set **CRC** because they both intersect **CSM**. The two-directional arrows indicate overlapping (intersecting) change sets.

Figure 8: Relationships between change sets.
← → Direct relationship
← – → Transitive relationship
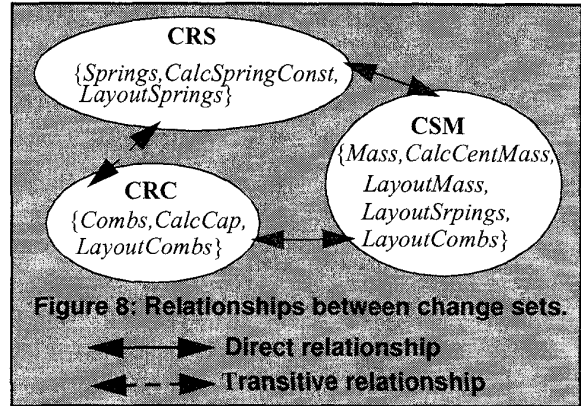
To relate those change sets which overlap directly or transitively, we define an *Affinity* relation (shown in *Figure 9*). By applying the *Affinity* relation, we analytically group change sets **CRS**, **CRC**, and **CSM**. The union of these sets guarantees that we will have no duplicate copies of the resulting data and operations. Now we encapsulate these data and operations into a software component. The resulting software component contains the data elements as well as operations on it. By applying the *Affinity* relation (an equivalence relation) across all of our change sets, we effectively partition the change sets. The equivalence class described above contains **CRS**, **CRC**, and **CSM**. Each equivalence class becomes a software component. The formal definition for this partition of change sets into software components follows in *Figure 9*.

$Overlap = \{(cs_1,cs_2) \lozenge CS \times CS | (cs_1 \cap cs_2 \neq \varnothing)\}$

$Affinity = \{(cs_1,cs_2) \lozenge CS \times CS |$
$\quad ((cs_1,cs_2) \in Overlap) \vee (\exists(cs_3 \lozenge CS)$
$\quad ;((cs_1,cs_3) \in Afffinity \wedge (cs_3,cs_2) \in Affinity))\}$

$SameComponent = \{(ec \subseteq CS) | (\forall cs_1,cs_2 \in CS$
$\quad ;\left(^{((cs_1 \in ec) \wedge}_{(cs_2 \in ec))} \Leftrightarrow ((cs_1,cs_2) \in Affinity)\right)\}$

Figure 9: Step 5 - Combine and componentize overlapping change sets.

Step 6: Add other necessary components.

Lastly in step six, we add software components to encapsulate those parts of the software solution not delineated in the previous steps. We need control logic to iterate the

basic steps of laying out the mass, combs, and springs and to simulate the sensitivity of the resulting accelerometer design to changes in capacitance of the combs. Good design principles dictate that a controller component not be cognizant of the internal details of the operations that it activates [19]. In addition, though the control logic is impacted by changes in the order by which small-effect operations are activated, an operation's implementation should not be affected by these changes. It is therefore reasonable for the control logic to be logically separate from the small-effect operations with respect to anticipated changes in the software solution or in its implementation.

Our analysis of CAD software for designing a MEMS accelerometer resulted in the software design components shown below in *Figure 10*. Each component has a boldface *change signature* which represents the anticipated changes associated with that component. Any encapsulated data is shown on the next line of the component diagram. The operations contained within the component are displayed in italics. We would make the expected changes either by replacing or by modifying only the software component associated with a particular change. As was discussed in step five, the mass, combs, and springs components are related transitively through the layout operations. The hashed line represents this grouping of the mass, combs, and springs sub-components into a larger component or package.
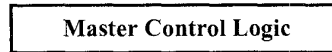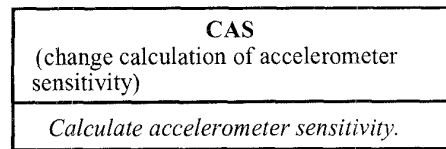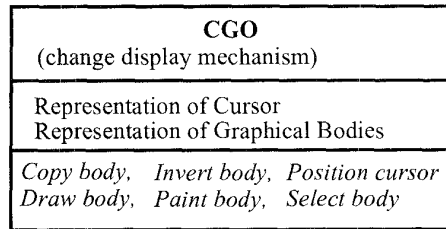

$CRC \cup CCC \cup CLC$
(change representation of combs, change calculation of capacitance, change layout of combs)

Representation of Combs

Calculate capacitance of combs
Layout Combs

$CSM \cup CCM \cup CLM \quad \cup CRS \cup CSAP \cup CSC \cup CLS$
$\cup CRC \cup CCC \cup CLC$

Basic MEMS Structures and Operations

$CSM \cup CCM \cup CLM$
(change shape of mass,
change calculation of center of mass,
change layout of mass)

Representation of Mass

Calculate center of mass
Layout mass

$CRS \cup CSAP \cup CSC \cup CLS$
(change representation of spring,
change spring anchor points,
change from table-driven to calculated
spring constant,
change layout of springs)

Representation of Spring

Calculate spring constant
Determine number of meanders
Layout springs

CIOS
(change mechanism for input/output of structural parameters)

Representation of Structural Parameters

Input/output structural parameters

CIOSS
(change mechanism for input/output of structural parameters)

Representation of Simulation Parameters

Input/output simulation parameters

CGO
(change display mechanism)

Representation of Cursor
Representation of Graphical Bodies

Copy body,   Invert body,   Position cursor
Draw body,   Paint body,   Select body

CAS
(change calculation of accelerometer sensitivity)

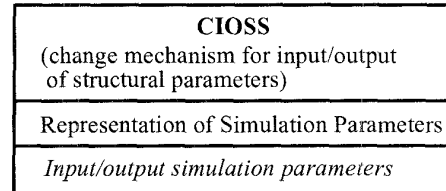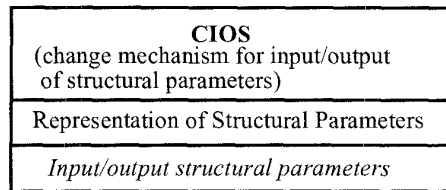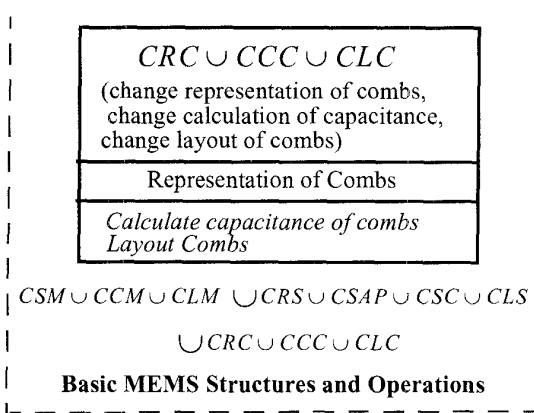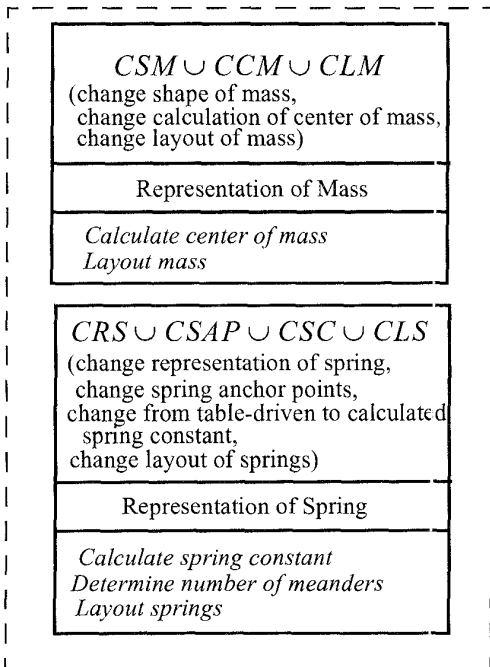*Calculate accelerometer sensitivity.*

Master Control Logic

Figure 10: Components resulting from the analysis of the MEMS design tool.

5. Partitioning Control Flow

In the previous section, we discussed the decomposition of large-effect operations into reusable small-effect operations and the partition of data and operations into components that localize anticipated changes to these components. In this section, we analyze change with respect to control flow. By control flow, we mean the order of execution of a set of tasks. For instance, in our MEMS design software example, the Master Controller activates a sequence of tasks $<t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_9,t_{10}>$ in which each task t_i is defined as presented below.

t_1 Input the structural parameters.

t_2 Layout mass.

t_3 Layout combs.

t_4 Layout springs.

t_5 Input simulation parameters.

t_6 Calculate sensitivity of the MEMS accelerometer.

t_7 Compare the calculated sensitivity to the required sensitivity.

t_8 Adjust layout of mass.

t_9 Adjust layout of combs.

t_{10} Adjust layout of springs.

Application requirements may vary over time thereby requiring changes to the original control flow. In some cases, the domain expert may specify a list of alternative activation sequences in the requirements specification. For instance, alternative sequences for the software design of the MEMS accelerometer are $<t_5,t_1,t_2,t_3,t_4,t_6,t_7,t_8,t_9,t_{10}>$, $<t_1,t_5,t_2,t_3,t_4,t_6,t_7,t_8,t_9,t_{10}>$, $<t_1,t_2,t_4,t_3,t_5,t_6,t_7,t_8,t_9,t_{10}>$, $<t_5,t_1,t_2,t_4,t_3,t_6,t_7,t_8,t_9,t_{10}>$, $<t_1,t_5,t_2,t_4,t_3,t_6,t_7,t_8,t_9,t_{10}>$, $<t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_{10},t_9>$, $<t_5,t_1,t_2,t_3,t_4,t_6,t_7,t_8,t_{10},t_9>$, $<t_1,t_5,t_3,t_4,t_5,t_6,t_7,t_8,t_{10},t_9>$, $<t_1,t_2,t_4,t_3,t_5,t_6,t_7,t_8,t_{10},t_9>$, $<t_5,t_1,t_2,t_4,t_3,t_6,t_7,t_8,t_{10},t_9>$, $<t_1,t_5,t_2,t_4,t_3,t_6,t_7,t_8,t_{10},t_9>$.

In our HASE'97 paper, we show that localizing invariant control sequences in separate control components heuristically reduces the complexity of making changes to the control flow [14]. The invariant control sequence for the master controller of the MEMS design software is $<t_6,t_7,t_8>$. The complexity of determining the invariant subsequences is of the order $O(m*n)$, where m is the number of alternative sequnces and n is the length of the required sequence.

We then partition the master controller into separate control flow components that localize the invariant subsequence as shown in *Figure 11*. A discussion of the consequences of this partition continues in the next section.
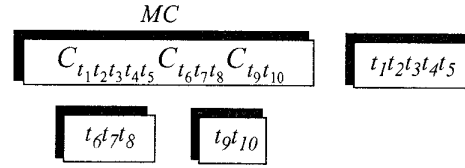

Figure 11: Control components which localize invariant subsequences.

6. Discussion of Results

Our analytical approach supports the fundamental design principles of decomposition, partitioning, and encapsulation as advocated for the modularization of complex systems [19]. One distinguishing feature about our approach is that the designer must critically consider what it means for an operation to be reusable. For instance, the operation of laying out the mass encompasses the reusable operation of calculating the center of mass as well as very small-effect operations which are trivial and therefore not reusable from our point of view. Likewise, decomposition of the large-effect layout of springs operation is necessary to better understand the details of this operation and to elicit reusable and smaller-effect operations such as the spring constant calculation. Here our method goes beyond our original approach in that we include not only the reusable smaller-effect operations but also the larger-effect operations that we think are reusable.

The other distinguishing feature of our approach is the systematic identification of feasible or anticipated changes to the solution and the analytical relationship of the data and operations that would be impacted by these changes. Our approach requires the designer to carefully specify the solution elements that would have to be modified to satisfy a particular change. This specification becomes the mathematical model for partitioning these solution elements into components. The algorithmic complexity of the partitioning process is polynomial.

Our approach complements existing methods by making more precise and systematic the process of partitioning a solution into components. Of note is the fact that the Booch analysis shown in *Figure 4* resulted in the abstractions of fingers for the combs and meanders for the springs. Our decomposition of the layout of combs and layout of springs operations also indicates that the basic combs and springs structural bodies consist of sub-bodies. The Booch method expects the designer to independently think of the useful abstractions, while our approach specifically directs the designer to separate the design of the "smaller bodies" from the design of the "bigger" bodies if the operations related to the smaller bodies are reusable.

Our analysis of feasible changes to the MEMS design tool solution helped us to identify those parts of the solution which fit into the randomized search algorithms that we studied previously [13]. The large component from *Figure 10* which encapsulates the mass, combs, springs, and layout operations is the population component of a genetic algorithm solution. Calculating the sensitivity of the accelerometer is the fitness or cost function, and the master controller provides the control logic as in the genetic optimization solution. This componentization will support the expected evolution of the MEMS design tools towards a more extensive use of optimization techniques for determining a good layout of accelerometers and other MEMS devices. As designers experiment with differently shaped structural components (masses, combs, etc.), the design space will become more complex. This complexity will motivate the application of randomized search techniques such as genetic algorithms which are suitable for encodable but complex search spaces [12].

In turn, the layout functions may merge into one operation for determining a new point (MEMS accelerometer layout) in the solution space. Likewise, the simulation of the sensitivity of the device may automatically iterate through a variety of simulation parameters. We would then replace the control flow components from *Figure 11* with a master controller consisting of one invariant sequence $<t_1,t_2,t_3,t_4>$ where t_1 is the input of the structural parameters, t_2 is the automated generation of a new layout, t_3 is the simulation of the accelerometer sensitivity, and t_4 is the comparison of the required to the calculated sensitivity. The master controller would loop to generate a new layout if the previously generated layout did not yield an acceptable sensitivity value. In the next section, we summarize our paper and discuss future research.

7. Summary and Future Research

The purpose of this paper was to demonstrate our approach for partitioning a software solution into components that reduce the impact of change thereby helping to preserve performance reliability and that support component reuse. First we discussed the role of partitioning in the transformation from requirements to design and showed via example that this process is highly dependent on the expertise of the human designer. The background section provided information about MEMS accelerometers and the software tools used to design them. We outlined our change analytic approach of grouping together data and operations that are impacted by the same changes and applied it to the synthesis of a high-level architecture for software tools to design MEMS components.

Three important features characterized our design approach: (1) the recursive decomposition of large-effect operations into small-effect operations, (2) the enumeration of anticipated or feasible changes to the software solution and the analytical grouping of solution elements impacted by the same changes, and (3) the identification of a heuristically good way to organize control flow elements. The process of decomposing for reuse and the identification of the impact of change are qualitatively dependent on human judgment: at present, the best we can do is to program the computer to remember our decomposition decisions and identification of changes. We can mathematically model and therefore automate the process of grouping together change dependent data and operations. Likewise, we can formally model and automate the identification of invariant subsequences in control flow sequences: a fact which leads us to an important observation and direction for our future work.

The representation of change dependencies as a set of related elements and control flow as a "sequence of symbols" is analogous to the work of information theorists who search for relationships between data in order to determine a "good" organization of the data. Similarly, we are researching ways to symbolically and mathematically represent what we know about "good" designs. Algorithms developed by researchers working in the areas of data analysis and clustering theory may help us to model and automate additional features of the design process. The reader should see [8] for an extensive collection of data clustering algorithms and [20] for a review of the literature on clustering theory. Though the idea of applying information-theoretical models to the analysis of software structural complexity is not new [4], our search of the literature shows that the systematic application of information theory to software design is an open area of study. We intend to further research and develop methods to systematically and semi-automatically synthesize designs that satisfy system design constraints such as evolvability and adaptability to available system resources.

8. Acknowledgments

9. References

[1]   Adams, J., and D. Thomas (1997), "Design Automation for Mixed Hardware-Software Systems", In *Electronic Design*, Vol. 45, No. 5, pp. 64-66, 71-72.

[2] Bass, L., P. Clements, and R. Kazman (1998-1), "A-7E: A Case Study in Utilizing Architectural Structures," Chapter 3 in *Software Architecture in Practice*, Addison-Wesley Publishing Company, Reading, MA, pp. 45-71.

[3] Bass, L., P. Clements, and R. Kazman (1998-2), "Analyzing Development Qualities at the Architectural Level," Chapter 9 in *Software Architecture in Practice*, Addison-Wesley Publishing Company, Reading, MA, pp. 189-220.

[4] Belady, L. (1981), "Complexity of Large Systems," Chapter 13 in *Software Metrics: An Analysis and Evaluation*, A. Perlis, F. Sayward, and M. Shaw, Eds., MIT Press, Cambridge, MA, pp. 225-233.

[5] Booch, G. (1994), *Object-Oriented Analysis and Design: With Applications*, Second Edition, The Benjamin/Cummings Publishing Company, Redwood City, CA.

[6] Britton, K., and D. Parnas (1981), *A-7E Software Module Guide*, NRL Memorandum Report 4702, Dec.

[7] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996), *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, Chichester, England.

[8] Diday, E. (1994), *New Approaches in Classification and Data Analysis*, Springer-Verlag, Berlin, Germany.

[9] Fedder, G. (1994), *Simulation of Microelectromechanical Systems*, Ph.D. Thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA.

[10] Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, Reading, MA.

[11] Gilbert, J. (1998), "Integrating CAD Tools for MEMS Design," In *Computer*, Vol. 31, No. 4, pp. 99-101.

[12] Goldberg, D. (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA.

[13] Hoover, C., and P. Khosla (1996), "An Analytical Approach to Change for the Design of Reusable Real-Time Software", In *Proceedings of the Second Workshop on Object-Oriented Real-Time Dependable Systems*, Feb. 1-2, IEEE Computer Society Press, Los Alamitos, CA.

[14] Hoover, C., and P. Khosla (1997), "Analytical Design of Evolutionary Control Flow Components," In *Proceedings of the Second Workshop on High Assurance Systems Engineering*, Aug. 11-12, IEEE Computer Society Press, Los Alamitos, CA.

[15] Koester, D., R. Mahadevan, and K. Markus (1994), *Multi-User MEMS Processes (MUMPs) Introduction and Design Rules*, MCNC MEMS Technology Applications Centers, 3021 Cornwallis Road, Research Triangle Park, NC 27709, Rev. 3, Oct.

[16] Lemkin, M. (1997), *Micro Accelerometer Design with Digital Feedback Control*, Ph.D. Thesis, Department of Mechanical Engineering, University of California at Berkeley, Berkeley, CA.

[17] MOSIS (1998), MOSIS VLSI Fabrication Service Price List, URL - http://www.mosis.org.

[18] Parnas, D. (1972), "On the Criteria To Be Used in Decomposing Systems into Modules," In *Communications of the ACM*, Vol. 15, No. 12, pp. 1053-1058.

[19] Parnas, D., P. Clements, and D. Weiss (1984), "The Modular Structure of Complex Systems," In *Proceedings of the Seventh International Conference on Software Engineering*, March, pp. 408-417, Reprinted in *IEEE Transactions on Software Engineering*, SE-11, pp. 259-266, 1985.

[20] Reinke, R. (1991), *Symbolic Clustering*, Ph.D. Dissertation, Report No. UIUCDCS-R-91-1704, Department of Computer Science, University of Illinois, Urbana-Champaign, IL.

[21] Selic, B., G. Gullekson, and P. Ward (1994), *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, NY.

[22] Shaw, M., and D. Garlan (1996), *Software Architecture: Perspectives on an Emerging Discipline*, Chapter 2: "Architectural Styles," Prentice Hall, Upper Saddle River, NJ, pp. 19-32.

[23] Ward, P., and S. Mellor (1985), *Structured Development for Real-Time Systems*, Vol. 2: Essential Modeling Techniques, Prentice Hall, Englewood Cliffs, NJ.

[24] Wortmann, J. (1996), *Object-Oriented Analysis for Advanced Flight Data Management*, Report No. 96-43, Technical University of Berlin, Berlin, Germany.

Appendix A: Impact of Feasible Changes

Change *c*		
	Change set *cs* of data and operations for change *c*	
Expected or Feasible changes	Data impacted by the change	Operations impacted by the change
CSM: Change the shape of the mass from a rectangle (as is currently used).	Mass (Add list of component shapes.)	Calculate center of mass. Layout the mass. Layout the combs. Layout the springs.
CRS: Change the representation of the springs.	Springs	Calculate spring constant. Layout the springs.
CRC: Change the representation of the combs.	Combs	Calculate the capacitance of the combs. Layout of combs.
CSAP: Change the number of spring anchor points.	Springs (Add list of anchor points.)	Calculate spring constant. Layout the springs.
CSC: Change from a table-driven to a calculated spring constant.		Calculate spring constant.
CCM: Replace calculation for the mid-point of a rectangular mass with calculation of the center of mass for a non-rectangular mass.		Add: Calculate center of mass.
CLM: Change the layout of the mass operation.		Layout the mass.
CLC: Change the layout of the combs operation.		Layout the combs.
CLS: Change the layout of the springs operation.		Layout the springs.
CCC: Change the calculation of the capacitance of the combs.		Calculate the capacitance of the combs.
ALS: Automate layout of mass, combs, and springs structures.		Replace layout of mass, layout of combs, and layout of springs with automated operation.
CIOS: Change mechanism for input/output of the structural parameters.		Input/output structural parameters.
CIOSS: Change mechanism for input/output of the parameters for simulating the sensitivity of the accelerometer.		Input/output simulation parameters.
CGO: Change graphical operations.	Cursor, Body	Position cursor; draw, paint, select, copy, and invert structural body.
CAS: Change calculation of the accelerometer sensitivity.		Calculate accelerometer sensitivity.