An Analytical Approach to Change for the Design of Reusable Real-Time Software

Carol L. Hoover and Pradeep K. Khosla
Carnegie Mellon University

Abstract

In this paper, we present an analytical method for incorporating knowledge about change into the design of reusable real-time software components. We apply this method to the construction of algorithmic software solutions that minimize the effect of anticipated changes in the solution. The motivation for our research is based on two premises: (1) software solutions that can easily be adapted to changes in the problem and solution domains are more readily reused and (2) the reuse of reliable software promotes the construction of high quality software systems and reduces development costs. This paper also briefly overviews our current research goals involving analysis and design methods to enable software engineers to rapidly adapt their real-time software solutions to changing application requirements.

Keywords: *Real-time software design, reusable software, change analysis.*

1 Background

It is well-known that real-time systems often have stringent timing and safety requirements [1]. The *ad hoc* way in which software engineers develop and maintain real-time software systems is often difficult, time-consuming, and costly [12]. To make matters worse, as in the example of the software problem that delayed the first NASA Space Shuttle orbital flight, the software process is sometimes error-prone. In discussing the Shuttle incident, Garman notes a key idea: though software is easy to change, it is also via change that software is the easiest part of a real-time system to compromise [3].

Software reuse, broadly defined as the use of engineering knowledge or artifacts from existing systems to build new ones, is considered to be an important technology for improving software quality, reducing development cost, and reducing the time-to-market [2].

Some researchers argue that reuse does not guarantee the safety of a software solution. For instance, Leveson states that safety is not just a property of the software itself but also a property of the software design and environment in which it is used [10]. It appears to be an open question as to whether or not it is easier to construct a safe software system totally from scratch or from parts of a "safe" existing software solution. As evidenced by the comprehensive literature dedicated to the topic of software reuse, it is clear that software experts consider software reuse to be an important technology [6, 8, 11].

The Chimera software infrastructure prescribes a standard format for building reusable software objects. These port-based objects automatically map to executable processes. The format includes operations for initializing, executing, and deactivating these dynamic objects as well as for error handling and inter-object communications [16]. The Chimera infrastructure includes a real-time kernel designed to ensure a performance-efficient and reliable real-time operating environment for robotics and factory applications [17].

Chimera objects are software packages that become processes activated by the real-time kernel. The concern of the Chimera infrastructure is with the *dynamic* and real-time world of the computer and its operating environment. In the *static* world of the human, software engineers design software solutions for real-time applications and partition these solutions into logical and eventually physical software components.

Static software components are mapped into dynamic processes in multiple ways and can ideally be executed in more than one operating environment. While Chimera addresses the reuse of dynamic process packages, we are concerned here with the reuse of static software components.

We would like to partition algorithms, such as those used to provide advanced motor control for robotic and factory automation applications, into software components that can be reused across various software solutions and if possible across different operating environments. A reusable software component is, for the purposes of our study, an executable or compilable module carefully designed to be useful in several programs, including unanticipated ones [18].

Before presenting our approach, we should mention that other researchers have also studied ways to decompose mathematical software solutions into software components. For instance, Westerberg's work reflects the common approach of intuitively decomposing algorithms into functional modules. Westerberg's modules can be used in various permutations to solve systems of linear equations [19].

We think our approach differs because we *systematically* decompose a software solution into small-effect operations. More importantly, we *analytically* recombine these small-effect operations into software components that localize the expected changes to the software solution.

2 Problem of Change

Concerned with the safety implications and the high cost of maintaining systems with stringent timing and reliability requirements, our goal is to study the nature of changes to software (*software metamorphosis*) and to develop analytical methods for designing and building real-time software solutions that can reliably and easily be changed.

From here on, we use the terminology "reduce the impact of change" to mean that a software solution can be modified in a reliable, timely, and cost-efficient manner. Our goal is to design reusable software components that *reduce the impact of making changes* to real-time software solutions.

From our experience in developing (and maintaining) industrial control software, we have found that changing a real-time software system can be tedious and time-consuming. Depending upon the type of change and the organization of the software, the software engineer may need to examine and modify multiple parts of the software system [5].

During the design of a new application, we would like to be able to experiment with alternative software solutions to achieve real-time performance or reliability. Sometimes we can achieve our goals via tunable parameters. Other times, we need to tune the solution by replacing poor-performing parts of the system [15].

We would like to localize the parts of a software system affected by a change because it is easier to modify software when the related changes are close together. Also, we are less likely to introduce errors into unrelated parts of the system if the unrelated part is logically or physically separate from the part affected by the change. In essence, we would like to examine and modify as few software components as possible.

We think it is reasonable that software engineers may decide not to reuse a whole or partial software solution if they perceive or discover that it would take more effort to modify an existing software solution than to create a new one. Therefore, we would like to find a systematic way to design reusable software components that localize change to as few software components as possible.

3 Analytical Approach

In this section, we propose target software properties and attributes that will serve as evaluation criteria for software resulting from our case study. We discuss the fundamental rationales for our analytical approach to building reusable software components that minimize the impact of change. Lastly, we outline the basic steps in our analytical approach and show how it applies to the construction of software components to implement optimization algorithms.

3.1 Target Software Properties and Attributes

Ideally, we would like to maximize a real-time software solution's capacity for change and to minimize the required engineering effort. So, our goal is to design software solutions that are *changeable* with respect to changing application requirements. But change to a software solution should preserve the predictability and reliability of the system.

In addition to the design of software components that enable flexible software solutions, we would like software solutions which are *partitionable* into concurrent processes. We would like to be able to easily distribute or replicate these processes across multiple processors if necessary to satisfy reliability and performance requirements.

Most importantly, we want to build software solutions whose components are *reusable* across various levels of granularity. This means that partial as well as whole solutions (software components as well as systems) can be used in alternative applications. We think the key is for software components to be used intact or replaced entirely if possible: modifying the internals of a software component requires substantial human effort and risks the introduction of error.

3.2 Fundamental Rationales

Two fundamental rationales direct our approach to designing reusable software components. One rationale is based on the idea of recasting large-effect algorithms into small-effect or partial algorithms. The idea is that software components that encapsulate whole algorithms more likely require internal modification in order to be reused; whereas, components that contain parts of an algorithm can be alternated to achieve variations in the algorithm's logic or implementation features [18].

Our interpretation of this rationale is that software solutions implemented as large-effect components are not likely to be as reusable as those implemented as small-effect components. One research question is how to recast a monolithic software solution into parts whose implementations will facilitate changes to the software solution via replaceable and reusable components.

The other rationale behind our approach is our idea that those parts of a software solution that are affected by the same expected changes should be located in the same software component. If we must replace or modify components to alter a software solution, we would like to minimize the number of affected components. Therefore, a second research question is how to design software components that minimize the impact of change.

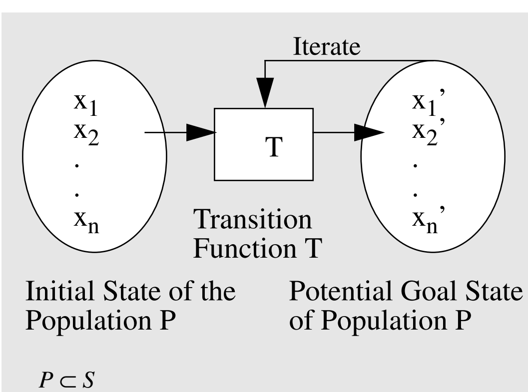
3.3 Detailed Approach

Our approach involves decomposition of the large-effect operations of a software solution and recomposition into software components based on an analysis of expected changes. Since many of our target software solutions involve experimental algorithms, we focus on decomposition at the algorithmic level and conduct a case study of three optimization algorithms.

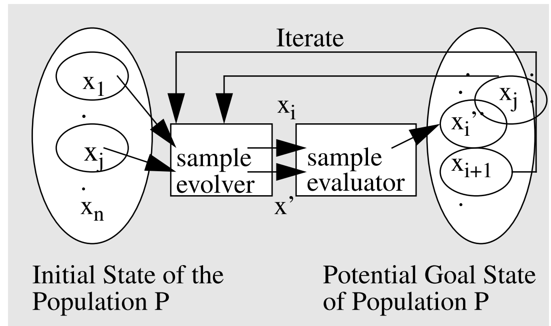
Our approach has the following five basic steps.

1. Identify the algorithmic features.
2. Decompose large-effect operations.
3. Determine the expected changes.
4. Componentize the operations using change analysis.
5. Add other necessary components.

In *step one*, we analyze the characteristic features of the algorithmic solution. For example, the characteristic features of a genetic algorithm (a type of optimization algorithm with which we are experimenting to select feasible control configurations) are an initial state of a population *P* of samples, a potential goal state of the population, and a transition function *T*. Note that samples x_i are members of a set *S* of all possible samples.



Step two is the decomposition of the large-effect transition function T into smaller-effect operations (a sample evolver and a sample evaluator). The interested reader should note that this version of a genetic algorithm always maintains a fixed-size population; therefore, a randomly generated new sample x' replaces a sample x_i to become x_i' only if it is more fit than x_i.



We continue the decomposition of the sample evolver into a randomized genetic operation selector and various genetic operations used to generate new samples. Likewise, we decompose the sample evaluator into a function to retrieve the saved fitness value of a particular sample, an objective function to calculate the fitness of a sample, and a comparator function to select the most fit of two samples. Further decomposition of these suboperations would result in smaller-effect operations which are trivial and therefore not significantly reusable.

We can now identify a set *O* of small-effect operations.

$$O = \{o \Diamond operator\}$$
$$= \{RandOperSel, GenOper, \ldots\}$$

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

Step three is to enumerate the types of changes that the researcher would expect to make to the software solution. These could be logical or algorithmic changes such as using a different genetic operator to generate a new sample or implementation changes such as changing the representation of a population member. We create a set C of expected changes.

$$C = \{c \Diamond change\}$$
$$= \{ChangeGenOper, ChangeSearchSpaceRep, \ldots\}$$

Step four involves the recomposition of the small-effect operations into software components based on the analysis of anticipated changes to the software solution. The goal is to group operations affected by the same changes into the same components. Then changes to the software solution or its implementation can be made simply by replacing whole components or by modifying a minimal number of components whose contents are affected by the changes.

Step four is a combination of several substeps. We relate each anticipated change to the software solution to the operations whose implementations would be affected by this change via a change impact relation *CI*.

$$CI \Diamond C \leftrightarrow O = \{(c,o) \Diamond C \times O | (c \rightarrow impact(o))\}$$

Applying the relation *CI*, we obtain a set of operations for each expected change. We will call these *change sets*. In totality, we now have a set of change sets *CS*.

$$CS = \left\{ cs \subseteq O \Big|^{(\exists c \Diamond C | (cs = CI(c))) \Lambda}_{(\forall (o \in cs), \exists ((c,o) \in CI))} \right\}$$

The size of a change set indicates the magnitude of the impact of the change associated with the set. For instance, modifying a particular genetic operator affects only the software implementation of this operator. The cardinality of the change set *CGO* associated with this change is one.

$$CGO = \{GenOper\}$$

Alternatively, changing the search space representation impacts the genetic operators, objective function, initialization of the population, and all other operations on population members. As we would expect, the cardinality of the change set *CSSR* associated with modifying the search space representation is larger than one.

$$CSSR = \{GenOper, ObjFunc, InitPop, \ldots\}$$

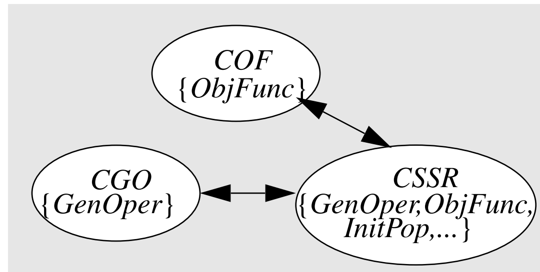
We might think that we are done in that each set of operations affected by a change could be the contents of a software component. Then in the future, all we would have to do to make the expected change is to exchange or modify the related software component. But such compositions could result in duplicate copies of operations. In the above example, both *CGO* and *CSSR* contain the genetic operations *GenOper*.

What is needed is a way to combine change sets that intersect one another. We define a relation called *Overlap* that associates two change sets that have a non-empty intersection.

$$Overlap \Diamond CS \leftrightarrow CS$$

$$= \{(cs_1, cs_2) \Diamond CS \times CS | (cs_1 \cap cs_2 \neq \emptyset)\}$$

We must also consider the case when change sets are related transitively through repeated application of the *Overlap* relation. For instance, in the diagram which follows, the change set *CGO* (operations affected by changing a genetic operation) is related to the change set *COF* (operations affected by changing the objective function) because they both intersect *CSSR* (operations affected by changing the search space representation). The two-directional arrows indicate overlapping (intersecting) change sets.



To relate those change sets which overlap directly or transitively, we define an *Affinity* relation.

$$Affinity \Diamond CS \leftrightarrow CS = \{(cs_1, cs_2) \Diamond CS \times CS |$$

$$\left. \begin{array}{c} ((cs_1, cs_2) \in Overlap) \vee (\exists (cs_3 \Diamond CS) \\ ;((cs_1, cs_3) \in Afffinity \wedge (cs_3, cs_2) \in Affinity)) \end{array} \right\}$$

By applying the *Affinity* relation, we analytically group change sets *CGO*, *COF*, and *CSSR*. The union of these sets ({*GenOper*, *ObjFunc*, *InitPop*,...}) guarantees that we will have no duplicate copies of the resulting operations. Now we encapsulate these operations into a software component. The resulting software component contains operations on population members. More will be said later about the interesting aspects of the results of this analysis.

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 *CGO*, *COF*, and *CSSR*. Each equivalence class becomes a software component. The formal definition for this partition of change sets into software components follows.

$$SameComponent = \{(ec \subseteq CS) | (\forall cs_1, cs_2 \in CS$$

$$\left. ; \left(\begin{array}{l} ((cs_1 \in ec) \Lambda \\ (cs_2 \in ec)) \end{array} \Leftrightarrow ((cs_1, cs_2) \in Affinity) \right) \right) \Big\}$$

Lastly in *step five*, we add software components to encapsulate those parts of the software solution not delineated in steps two through four. In the case of our genetic algorithm, the iterator logic that directs the detailed steps (e.g. sequence of function calls) of the algorithm can be encapsulated in a controller component.

Good design principles dictate that a controller component not be cognizant of the internal details of the operations that it activates. 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. Likewise, we should organize the control logic as a separate software component.

4 Application and Results

Our analysis of a genetic algorithm resulted in the six basic software components shown below. Each component has a boldface *change signature* which represents the anticipated changes associated with that component. The operations contained within the component are shown in italics. We were able to make the expected changes either by replacing or by modifying only the software component associated with a particular change.

CROS (Change Randomized Operator Selector)
Randomized Operator Selector

CCP (Change Control Parameters)
Input/Output Control Parameters

CVBIS (Change Value-Based Item Selector)
Value-Based Item Selector

CMC (Change Master Control Logic)
Genetic Master Control

CGO *U* CSSR *U* COF (Change Genetic Operator, Change Search Space Representation, Change Objective Function)
Representation of Population
Genetic Operators *Objective Function* *Initialize Population* *Output Population* *Retrieve Population Member* *Update Population Member*

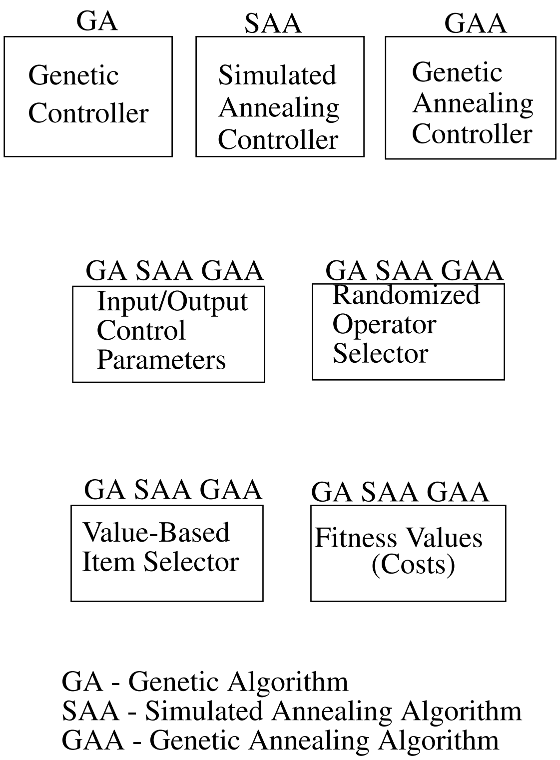
CFVR (Change Fitness Value Representation)
Representation of Fitness Values
Initialize Fitness Values *Output Fitness Values* *Retrieve Fitness Value* *Update Fitness Value*

We also applied our analysis technique to two other optimization algorithms: a simulated annealing algorithm and a genetic annealing algorithm which is a hybrid algorithm having characteristics of both a simulated annealing algorithm and a genetic algorithm.

Interestingly, we found that we could reuse the software components across the three different optimization algorithms. The primary difference occurred in the data format used to represent the solution space being optimized. The genetic and hybrid genetic annealing algorithms operate on populations of samples and can share the population software component. The simulated annealing algorithm optimizes a configuration.

As we would expect, each optimization algorithm has different control logic and therefore requires its own controller software component. Shown below are the software components resulting from our analysis. We indicate the extent to which each component could be reused across the three algorithms.

GA GAA	SAA
Population	Configuration



5 Conclusions

In this section, we experientially and rationally explain how the software components resulting from the application of our analytical approach satisfy the software evaluation criteria discussed earlier. We relate the results to the design of objects and also discuss the potential for automating the technique.

We restate our goal to build software components that reduce the impact of making changes to software solutions, that promote software reuse, and that further the potential for concurrency and parallelization. Equally important is the constraint that we must be able to make changes to software solutions while preserving the predictability and reliability of the existing solutions.

By localizing the parts of a software solution affected by a particular change, we found it easier to make experimental changes to a componentized software solution than to a monolithic software solution (all parts of the solution in one component). For instance, we could easily change the representation of the population by modifying or replacing only the population component.

In the monolithic solution, we had to locate and modify those sections that operated directly on population members. We could more quickly and easily make the necessary changes in a smaller, more functionally coherent population component because there was less distance between related changes than in the monolithic program component.

By reusing those parts of the solution not impacted by a change, we could guarantee that the unaffected parts perform as predictably and reliably as before the change was made. But in the monolithic program component, we had to be careful not to introduce errors into unrelated sections of the monolithic solution.

From a real-time point-of-view, the ability to make localized changes by replacing components is valuable [15]. To improve the performance of a genetic operator, we modify or replace only the population component. As expected, we found that our decomposed software solution (in contrast to our monolithic solution) could be executed as concurrent processes. We found that it would take substantial manual labor to separate the monolithic solution so that it could be executed in multiple processes: the only feasible scenario was to execute it as a single process.

We were easily able to execute our small-effect, reusable components as concurrent processes. In one mapping from reusable software components to processes, we packaged the input/output component in one Chimera process and the other components together in another Chimera process. Other mappings to Chimera processes are certainly possible.

The interesting point is that had we followed an intuitional partitioning with sole attention to abstract data types (ADT's), we may not have organized the input and output of the control parameters as a separate component. Though the control parameters could have been encapsulated as an ADT, actual input and output of these parameters may have been intertwined with the control logic. Experience or engineering guidelines may have directed us to separate the input/output logic from the master control logic [13], but even well-known design principles are not always applied systematically.

In our monolithic optimization programs the input/output operations and control logic were integrated and could not easily be mapped into separate processes. But by analyzing the expected changes, we thought of changing the facility for the input and output of control parameters. Our analysis *systematically* directed us to create a separate component for these operations.

Our analytical approach supports the fundamental design principles of decomposition, partitioning, and encapsulation [14]. The population and cost components encapsulate data objects similar to those that would result from an object-oriented analysis. Our approach also gave us direction about how to organize non-data parts of the software solution such as the randomized operator selector and the master control logic. While the definition of objects still depends largely on the creative thinking and intuition of the human, our components originate from a systematic application of specific changes that we expect to make to software solutions.

We have outlined in detail the mathematical expressions for step four of our analytical approach to demonstrate its potential for automation. Identifying the algorithmic features, decomposing the large-effect operations, determining the expected changes and their impact on small-effect operations, as well as adding control logic components currently require significant human thought.

6 Future Research

In this case study, the results of analytically applying knowledge about change to the construction of software components was valuable: our change analytic approach systematically guided us to partition our three optimization algorithms into software components that minimized the impact of change. We think further application to software solutions expressed as concise algorithms would yield similar results. We plan to test this hypothesis using real-time applications such as feed-back and control and digital filters.

We also note that in our study change analysis helped us to partition the software solution into reusable entities. Further study is needed to determine the precise relationship between change and the granularity level of reuse. Clearly, software solutions which are difficult to change are less likely to be reused (unless they can be used intact). But do software components organized on the basis of localizing change necessarily lead to the construction of reusable components and software solutions?

Jackson and Jackson suggest that hierarchical decompositions yield reusable software modules in numerical applications but not necessary so in other domains [7]. Would our approach yield reusable components in less mathematically-oriented real-time applications such as multi-media and real-time financial data reporting?

Another problem that we need to research is the possibility that components organized according to anticipated changes may exhibit undesirable software qualities and discourage reuse. For instance, is it possible that a software component could become too large or complex because many change sets of operations are transitively related (the *Affinity* relation in step four)? Could this size or complexity discourage reuse of the component?

Decomposing software solutions and relating anticipated changes to the decomposition solution is still vastly a human effort. We plan to study the types of changes that software engineers make to real-time software systems and to develop a model for categorizing change and relating change to software solutions during the design phase.

Our overall goal is to guide the software engineer in the design of reusable software components that facilitate cost-effective and risk-reduced changes to real-time software systems. We need models for building knowledge bases of change and for incorporating information about change into the design process. Similar to the type of analysis used to estimate the dependability of MARS application designs [9], we would like to be able to estimate the cost of software change. Halang states that real-time software systems are more likely to be predictable and dependable if they are simple [4]. Our analytical techniques should seek to reduce software complexity.

We think there is a need to formally study the way in which software engineers think about change during the design of new software components and about reuse when building new software solutions. Some research questions to be answered are the following.

- How should software engineers think when designing software solutions from reusable components?
- How can software engineers better anticipate and plan for change?

References

1. A. Burns and A. Wellings, *Real-Time Systems and Their Programming Languages*, Addison-Wesley Publ., Wokingham, England, 1990, pp. 2-10.
2. W. B. Frakes and S. Isoda, “Success Factors for Systematic Reuse,” *IEEE Software*, Sept. 1994, pp. 14-19.
3. J. R. Garman, “The ‘Bug’ Heard ‘Round the World,” *ACM SIGSOFT: Software Engineering Notes*, Vol. 6, No. 5, Oct. 1981.
4. W. A. Halang, “Real-Time Systems: Another Perspective,” K.M. Kavi, ed., *Real-Time Systems: Abstractions, Languages, and Design Methodologies*, IEEE Computer Society Press, Los Alamitos, Ca., 1992, pp. 11-18.x
5. C. L. Hoover, “The Role of the Real-Time Software Engineer: An Introductory Course,” in *Proc. of the Eighth SEI Conference on Software Engineering Education*, New Orleans, La., Mar./Apr. 1995, Springer-Verlag, Berlin, pp. 167-186.
6. *IEEE Software* (issue featuring systematic reuse), Sept. 1994.
7. D. Jackson and M. Jackson, “Problem Decomposition for Reuse,” Technical Report: CMU-CS-95-108, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., 1995.
8. T. Capers Jones, “Reusability in Programming: A Survey of the State of the Art,” *IEEE Trans. on Software Engineering*, Vol. 10, No. 5, Sept. 1984, pp. 488-494.
9. H. Kopetz et. al., “Distributed Fault-Tolerant Real-Time Systems: The Mars Approach,” *IEEE Micro*, Vol. 9, No. 1, Feb. 1989, pp. 25-40.
10. N. G. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley Publ., Reading, Mass., 1995, pp. 30-31.

11. H. Mili, F. Mili, and A. Mili, "Reusing Software: Issues and Research Directions," *IEEE Trans. on Software Engineering*, Vol. 21, No. 6, June 1995, pp. 528-562.
12. A. K. Mok, "Towards Mechanization of Real-Time System Design," A.M. van Tilborg and G.M. Koob, eds., *Foundations of Real-Time Computing: Formal Specifications and Methods*, Kluwer Academic Publ., Boston, 1991, pp. 1-37.
13. D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, Vol. 15, No. 12, Dec. 1972, pp. 1053-1058.
14. D. L. Parnas, P.C. Clements, and D. M. Weiss, "The Modular Structure of Complex Systems," *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 3, Mar. 1985, pp. 259-266.
15. M. Sitaraman, "Performance-Parameterized Reusable Software Components," *Int'l Journal of Software Engineering and Knowledge Engineering*, Vol. 2, No. 4, 1992, pp. 567-587.
16. D. B. Stewart and P. K. Khosla, "The Chimera Methodology: Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects," in *Proc. of the IEEE Workshop on Object-Oriented Real-Time Dependable Systems* (WORDS'94), Dana Point, Ca., Oct. 1994.
17. D. B. Stewart, D. E. Schmitz, and P. K. Khosla, "The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications," *IEEE Trans. on Systems, Man, and Cybernetics*, Nov./Dec. 1992, pp. 1282-1295.
18. B. W. Weide, W. F. Ogden, and M. Sitaraman, "Recasting Algorithms to Encourage Reuse," *IEEE Software*, Sept. 1994, pp. 80-88.
19. K. Westerberg, "Development of Software for Solving Systems of Linear Equations," Technical Report: EDRC-05-35-89, Engineering and Design Research Center, Carnegie Mellon University, Pittsburgh, Pa., 1989.