Title
Metric-driven reengineering for static concurrency analysis

Authors
Levine, David L.
Taylor, Richard N.

Publication Date
1993-02-08

Peer reviewed

Metric-Driven Reengineering for Static Concurrency Analysis

David L. Levine
Richard N. Taylor

levine@ics.uci.edu, taylor@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine* 92717-3425

UCI–ICS Technical Report TR-93-4
Arcadia Technical Report UCI-93-01

February 8, 1993

Abstract

An approach to statically analyzing a concurrent program not suited for analysis is described. The program is reengineered to reduce the complexity of concurrency-related activities, thereby reducing the concurrency state space. The key to the reengineering process is a metric set that characterizes program task interaction complexity and provides guidance for restructuring. An initial version of a metric set is proposed and applied to two examples to demonstrate the utility of the reengineering-for-analysis process. The reengineering has potential benefits apart from supporting analyzability, following the dictum that if it is hard to analyze, it is hard to understand and maintain.

*This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grant Number MDA972-91-J-1010. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Contents

1 Introduction **1**

2 Research Methodology **2**

3 The Reengineering-for-Analysis Process **3**

4 Metric Set Development **8**
- 4.1 Estimated Concurrency Graph Size 8
- 4.2 Nondeterminism Metrics 9
- 4.3 Graph Theoretic Metric 12
- 4.4 Information Flow Metric 13

5 Metric Evaluation **15**

6 Examples **17**
- 6.1 Elevator 18
- 6.2 Gas Station 21

7 Related Work **22**

8 Conclusion **23**

List of Tables

I C_1 Reflects `accept` Statement Nondeterminism 12
II Correlation of Complexity Metrics to Maintenance-Related Qualities . . 13
III Concurrent Program Complexity Evaluation Properties 16
IV Metric Evaluation Summary 16
V Metric Scores for Two Versions of Elevator Example 19
VI Metric Scores for Two Versions of Gas Station Example 21

List of Figures

1 Reengineering-for-Analysis Data Flow 5
2 Development of Metric Set and Restructuring Rules 6
3 Program with Two Kinds of Nondeterminism 11
4 Elevator Task Structure Diagram 18
5 Elevator Task and Program Metrics 20

1 Introduction

Analysis of concurrent programs is notoriously problematic. In addition to the difficulties of sequential program analysis, nondeterminism and scheduler dependencies mean that execution results depend on more than just program inputs: a concurrent program can follow one of many distinct possible execution paths for a given set of input data. Enumeration of all of these paths, as performed by reachability analysis, is not feasible for large-scale programs (or many small programs). Compositional analysis approaches offer relief from this state-space explosion problem, but only for certain program structures.

If it can be completed with available resources, static analysis provides valuable assurances of freedom from undesirable conditions such as deadlock and shared variable access anomalies, and eventual progress to specified states or events. Given a program that is too large to statically analyze with available technology, how can we attain these benefits? One approach is to reengineer the program such that it meets the original specification, yet has a task structure that is more amenable to analysis. We propose a metric-driven reengineering process in which the program tasking structure is iteratively evaluated and restructured.

This document contains an overview of the approach to reengineering for analysis, with a primary focus on metrics used to evaluate the program and guide its restructuring. The following two sections describe the research methodology and the reengineering-for-analysis process. The set of metrics currently used by the process is developed in Section 4 and evaluated in Section 5. Highlights of the application of the reengineering-for-analysis approach to two examples are reviewed in Section 6. Related work in the areas of structural metrics and restructuring concurrent programs is overviewed in Section 7. Section 8 concludes with some thoughts on future directions for this research. The ideas presented here are not specific to Ada programs, in general, though Ada terminology is used for convenience.

2 Research Methodology

While it would be most desirable to design programs with analyzability as a design goal, this is typically not pursued because of the lack of support by popular design methodologies. The long-term goal of this research is to augment these methodologies with methods for enhancing analyzability. It is based on the experiences with static analysis of various programs that has provided insight of value for analysis of other, general programs. An intermediate step towards the long-term goal is to enhance the analyzability of existing programs through application of a well-defined reengineering process.

The research plan starts with definition of analyzability and expression of a mechanism for relating program structure to analyzability. The bulk of the research involves demonstrating the validity of this mechanism and its application to enhancing the analyzability of candidate programs. In concrete terms, the mechanism for relating structure to analyzability is structural complexity metrics. Hypotheses are advanced regarding their application to characterizing and enhancing analyzability and attempt to validate them by way of experiment on a series of successively more complex, real-world example programs.

A narrow definition of analyzability reflects the empirical origins of this research and helps focus the direction, at least for the near term: analyzability is defined as the size of the concurrency state space of a program. Two other facets of analyzability that we plan to explore, though not in this report, are the number of spurious error reports generated during analysis, and a measure of the compositionality of analysis results on separate system components. Spurious error reports occur due to data folding; one straightforward technique for ameliorating this is to unroll variables that affect the synchronization-related control flow and that can assume only a limited range of values[1]. Compositional analysis approaches based on process algebra may be used to control state explosion [YY91], though applicability is limited to hierarchically structured systems. A long term subgoal of this research is to show how such structures can be characterized, and how (re-)design can target them.

[1] This can reduce the concurrency graph size, as well.

The key to the reengineering process is a metric set that characterizes program task interaction complexity and provides guidance for restructuring. Two of the key insights will be expressed as hypotheses, and through this research I will attempt their validation. The hypotheses are:

Hypothesis 1. Metric set scores can predict analyzability.

Hypothesis 2. Metric set scores can identify the structural design features to modify in order to improve analyzability.

These hypotheses are weak: their validity can be demonstrated by successful application to a single program. However, they will be applied to several non-trivial, real-world programs in order to illustrate their utility. The hypotheses are designed to serve as a starting point for metric-driven analysis-through-reengineering approaches.

The validation will be theoretical as well as empirical. The theoretical component utilizes existing metric evaluation criteria, such as those of Weyuker, for evaluating syntactic software complexity measures [Wey88]. The empirical component relies on experiments consisting of instantiation of the reengineering-for-analysis process and application to several examples.

The following sections reflect our current views of the two key components of the reengineering-for-analysis process. The first is a static description of the process itself, while the other is the initial metric set.

3 The Reengineering-for-Analysis Process

A well-defined reengineering process directs the evaluation and restructuring of a concurrent program for analysis. The goal of using the process is to transform an arbitrary concurrent program into one that can be statically analyzed with the current technology. Existing concurrent program design methodologies do not, in general, address analyzability; therefore, typical programs are often amenable to improvement in this regard.

Current analysis technology, though ever improving, will for the foreseeable future limit our opportunity for achieving the goal of analyzing arbitrary programs. We scale

this back to the goal of handling non-pathological programs that can be translated to a convenient internal representation.

Reengineering Before discussing the approach itself, a brief discussion of why it is termed *reengineering* helps clarify our direction. Chikofsky and Cross classify terms describing reengineering and related activities based on whether the activity occurs within or between abstraction levels, which are stages of a life-cycle development model [CC90]. Reengineering is defined as "examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form". It includes, for the purpose at hand, *reverse engineering* and *restructuring*. Reverse engineering is the analysis of a system in order to identify its components and their interrelationships, and the creation of representations of these. It spans the design and implementation abstraction levels. Restructuring is a transformation entirely within the design level. The functionality and behavior specified by the requirements must be preserved.

The proposed reengineering approach starts with an evaluation of the tasking structure of the program. Then, a structural redesign transforms the program to another that is more readily analyzed. A followup evaluation indicates whether the improvement in analyzability is sufficient; the restructuring/evaluation steps are repeated as necessary.

The reengineering process data flow is shown in Figure 1. Ovals represent process inputs and outputs, labeled arcs are internal data items, and rectangles signify process steps. The program is first translated to an internal (task interaction graph, or TIG) representation, briefly described in Section 4. Conceptually, restructuring is performed on the TIGs, as shown in the figure, because they contain all (and only) the concurrency-related program semantics. In practice, it is much more convenient to manipulate design representations and program source code. Translation to and from TIGs is mechanical, and therefore restructuring is typically performed on the more familiar representations.

In addition to the concurrent program to be analyzed, inputs to the reengineering and restructuring steps include an evaluation metric set and restructuring rules and guidelines. These are generated and refined as experience accrues with application of the reengineering for analysis process. The metric set development process, Figure 2, is the main focus

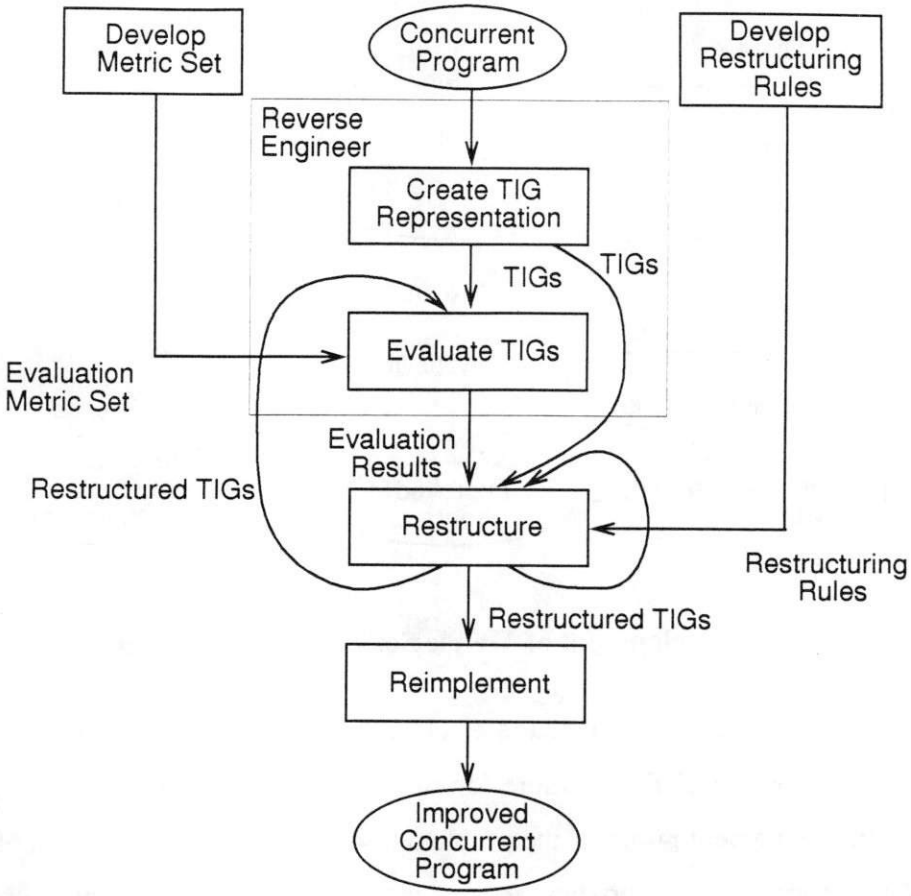

Figure 1: **Reengineering-for-Analysis Data Flow**

of this paper. Each metric serves one of two purposes, either evaluating the program analyzability or driving its restructuring. The restructuring rules encode knowledge of analyzable concurrent program paradigms. The input programs may be either contrived examples or actual programs to be analyzed. The former are used for metric validation and calibration while the latter are used for metric refinement; the same process is used for these two purposes.

Metric set refinement and adjustment will be loosely based on an analogy to perturbation theory. This is often used for approximating the solutions to differential equations and approximation of integrals by way of formulas as opposed to, for example, closed-form solutions or approximations by numerical analysis [Mur91]. The formulas are used

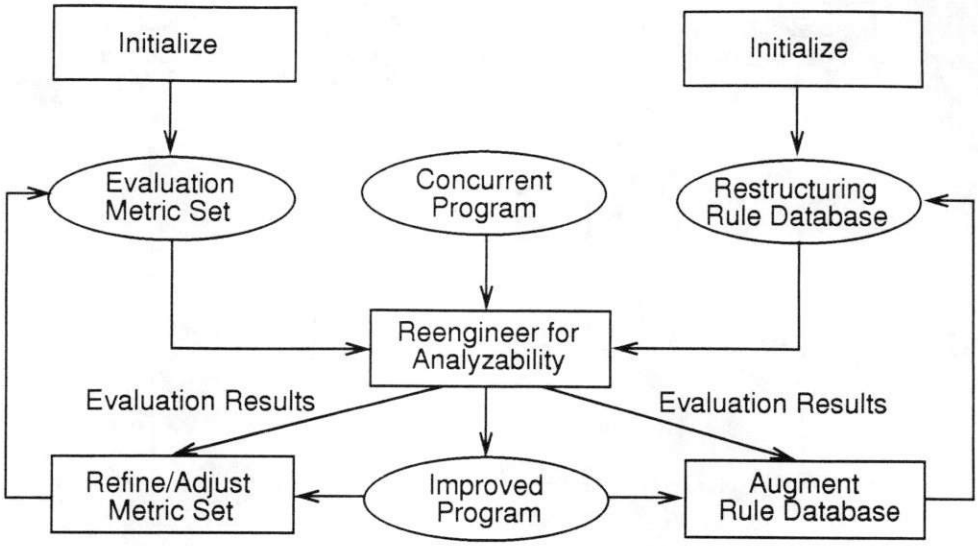

Figure 2: **Development of Metric Set and Restructuring Rules**

to calculate solutions to problems that are "close to" the original problem, followed by successive perturbation of the solvable formulas toward an acceptable approximation. For the metric refinement purpose, the problem to be solved is quantification of particular structural program characteristics. Analogues to the solvable formulas are metric set calculations for individual examples of each simple structure. Structural modifications to these simple examples represent the perturbations. Section 4.2 includes one case study.

Metric set adjustment includes the addition or deletion of individual metrics in order to capture the effects of specific structural modifications. Metric set refinement utilizing the perturbation approach ensures that metrics respond to structural features, and furthermore that they respond in the proper direction. Interactions of several structural features, in terms of effects on the metrics, can be explored by comparing the combined effects of multiple modifications with their effects in isolation.

Restructuring At this early stage of development of reengineering for analyzability, no attempt is made to automate the restructuring transformation; it is viewed as an instantiation of a design methodology, a creative process. To guide the designer, metric-based restructuring rules and guidelines will be developed. This should help the designer restrict

the target design to one that is analyzable. This strategy may be viewed as an analogue to structured programming for sequential programs: there is a small set of well known building blocks, the members of which can be readily reasoned about. And, this small set is sufficient to construct any program structure.

There are two relevant notions of structure: the external structure of task interactions and the internal structure of each task. External structure identifies groups of tasks that can be treated as separate analysis units, while internal structure captures the effects of features known to hinder analysis. Analogous features from sequential programs are `go to` statements and global variables. Examples of tasking features that complicate analysis are dynamic task creation and destruction and `delay` statements (if they affect the sequence of task interactions). Restructuring rules and guidelines are not considered further in this paper.

The criterion for judging success of the reengineering is whether the estimated size of the program concurrency graph will be smaller than the largest that is known to be manageable with the target analysis tools. The concurrency graph is a reachability graph in which each node represents a unique concurrency state, which is a tuple of states, for each task, with regard to synchronization activity. Each edge in the concurrency graph represents a single synchronization-related action.

Program Understandability Understandability is mentioned in several places in this paper. Structural features that affect analyzability often affect program understanding. Concurrent programs are difficult to understand, possibly for the same reasons that they are difficult to analyze. In particular, results of a concurrent program execution depend on more than just program inputs: nondeterminism must be considered, such as through relative task execution speeds and run-time scheduler design. The concurrency state space is typically quite large, *e.g.*, hundreds of thousands of states for a program with just a handful of tasks. The complexity metrics should provide insight into understandability by quantifying it: the complexity scores can be compared to those of program paradigm structures that are well understood and known to be analyzable, based on the size of the concurrency state space.

4 Metric Set Development

An evaluation of the analyzability of the original design is performed in order to characterize its structure. A set of metrics based on program tasking structures and features is being developed to implement this characterization. Structural complexity metrics for evaluating non-concurrency related characteristics have been developed and evaluated by many researchers, including [HMKD82]. These may be categorized based on whether they characterize program size, control flow, data flow, or information flow. Our set is initialized with metrics analogous to these, but for concurrent programs. This set includes four groups of metrics: one for estimating concurrency graph size, the second for evaluating the effect of nondeterminism on concurrency, the third based on the graph theoretic cyclomatic complexity, and the last based on information flow.

Several of the metrics are based on analysis of the task interaction graph representation (TIG) of the program. TIGs fold all non-tasking related activity, *i.e.*, sequential code regions, into nodes of a graph [LC89]. TIG edges represent task interaction actions, *e.g.*, synchronization, task creation or termination, and shared variable access. The concurrency graph that can be constructed from the program TIGs is called a task interaction concurrency graph (TICG).

A prototype tool has been constructed that calculates the values of each of the metrics from the TIG task representations. The TIGs are generated mechanically from program source code, thus the metric calculations are completely automated. Example output from the tool is shown in Figure 5, discussed in Section 6.1.

4.1 Estimated Concurrency Graph Size

The foremost obstacle to static concurrency analysis is state-space explosion. The metric set contains one metric that roughly estimates the state-space size for a given program by capturing the the exponential character of its growth. Its purpose is primarily to evaluate the analyzability of a program.

Wampler found that for the dining philosophers program, the number of concurrency graph states is approximately equal to $\left(\frac{N}{T}\right)^{\frac{T}{2}}$ where N = total number of state nodes in

all tasks, T = total number of tasks in the system, and therefore $\frac{N}{T}$ is the average number of state nodes per task [Wam85]. This metric is denoted W in the remainder of this paper.

Concurrency graph size can be estimated using the approach presented in [TLK92]. In that approach, the counts of state nodes for each task in the program are calculated from the program flowgraph or estimated based on tasking features in the program source code. These counts are inputs to Wampler's empirical formula for estimating concurrency graph size. Note that the state node count is roughly the same as the number of TIG nodes, the difference being that TIG nodes are duplicated for loops because a node can have at most a single entrance edge.

4.2 Nondeterminism Metrics

Shatz proposed a metric for the total complexity of a distributed program as a function of the local complexity of each task and the communication complexity, CC, due to task interactions [Sha88]. A suggested measure for CC was the count of communication statements. Later work by Damerla and Shatz refined the role of nondeterminism on CC. They offered empirical evidence that potential nondeterminism adversely affects understandability, with real nondeterminism further affecting it. They developed two metrics to quantify the nondeterminism associated with Ada `accept` statements (entry queue order) and with `select` statements [DS92]. The former are summed over the e task entries in all program tasks that are not in `select` statements:

$$\alpha = \sum_{i=1}^{e}(Calls_i - 1)$$

where $Calls_i$ is the number of calls on entry i, and

$$\alpha' = \sum_{i=1}^{e}(\prod_{j=1}^{T_i} x_{ij} \cdot (2^{T_i} - (T_i + 1)))$$

where T_i is the number of tasks with calls on entry i and
x_{ij} is the number of entry calls in task j on entry i,

while the latter are summed over the s `select` statements in all program tasks:

$$\beta = \sum_{i=1}^{s}(Calls_i - 1)$$

where $Calls_s$ is the number of calls on all entries in `select` statement i, and

$$\beta' = \sum_{i=1}^{s}(\prod_{j=1}^{T_i} y_{ij} \cdot (2^{T_i} - (T_i + 1)))$$

where T_i is the number of tasks with calls on `select` statement i and y_{ij} is the number of entry calls in task j on all entries in `select` statement i.

There are two groups of metrics for each, denoted without and with primes. The first (unprimed) is simply the total number of callers for each entry, or each entry in the `select` statement. The second (primed) incorporates the "clustering" and "spreading" of entry calls among program tasks in an attempt to account for the difference between potential and real nondeterminism. A trivial example of the difference is that if two entry calls are clustered in one task, there is no real nondeterminism between them. But if the calls are spread across two different tasks, then there could be. Finally, the sums of corresponding metrics above are defined:

$$\gamma = \alpha + \beta$$

and

$$\gamma' = \alpha' + \beta'.$$

A short case study to demonstrate the metric evaluation and adjustment process was performed using the simple program in Figure 3. The three *C* tasks each call one of the entries in the *S* task. This example exhibits the two kinds of nondeterminism captured by α (entry queue ordering) and β (choice of `select` statement alternatives). The α, β, and γ values for this program are, respectively, 0, 2, and 2. A perturbation is introduced by replacing the entry call in task *C3* to *S.E2* with a call to entry *S.E1*. The potential nondeterminism associated with the `accept` *E1* statement increases because of the additional caller; however, the α, β, and therefore γ values remain unchanged.

A deficiency with the α and α' metrics is apparent: they do not include entries that appear in `select` statements. Therefore, the nondeterminism of entry queue ordering on these entries is not reflected in the metrics. We modify them slightly by including these entries in calculations of α and α', and therefore γ and γ':

$$C_1 = \sum_{i=1}^{e+s}(Calls_i - 1) + \beta$$

```
procedure NondeterminismExample is

    task S is
        entry E1;
        entry E2;
    end S;

    task C1;
    task C2;
    task C3;

    task body S is
    begin
        loop select
            accept E1;
        or
            accept E2;
        end select; end loop;
    end S;

    task body C1 is
    begin
        S.E1;
    end C1;

    task body C2 is
    begin
        S.E1;
    end C2;

    task body C3 is
    begin
        S.E2;          -- replace with S.E1 to increase load on that entry queue
    end C3;

begin
    null;
end NondeterminismExample;
```

Figure 3: **Program with Two Kinds of Nondeterminism**

where $Calls_i$ is the number of calls on entry i, and

$$C_1' = \sum_{i=1}^{e+s} \left(\prod_{j=1}^{T_i} x_{ij} \cdot (2^{T_i} - (T_i + 1)) \right) + \beta'$$

where T_i is the number of tasks with calls on entry or `select` statement i and x_{ij} is the number of entry calls in task j on entry i.

C_1 characterizes the nondeterminism in a program due to the presence of task entry queues and `select` statements in Ada programs. In addition, C_1' incorporates the degree of spreading of callers across tasks to eliminate the effects of some potential nondeterminism that can not be realized. C_1 reflects the increase in potential nondeterminism with the perturbation of the example in Figure 3, as shown in Table I. C_1 and C_1' serve two purposes: they predict concurrency graph size[2] and therefore analyzability, and they will be used to guide restructuring.

Table I: **C_1 Reflects `accept` Statement Nondeterminism**

program version	α	β	γ	C_1
baseline program, with both `accept` and `select` statement nondeterminism	0	2	2	3
perturbed program, with increased `accept` statement nondeterminism	0	2	2	4

4.3 Graph Theoretic Metric

A metric analogous to McCabe's complexity measure [McC76] is based on the cyclomatic number of a graph. Sellers advocates the formulation $V(G) = E - N + P + 1$, where $V(G)$ is the cyclomatic complexity of a program control flow graph and E, N, and P are the number of edges, nodes, and connected components, respectively, in the graph [Sel92]. For sequential programs, P is the number of modules[3]; for concurrent programs, P naturally represents the number of tasks, so:

$$C_2 = E - N + T + 1$$

where E is the total number of edges in all of the program TIGs, and N is the total number of nodes in all of the program TIGs.

[2] `accept` statement nondeterminism shows up in the depth of the concurrency graph while `select` statement nondeterminism appears as fan-out of a concurrency graph state node.

[3] Sellers showed that modularization of a program does not increase this $V(G)$, though, undesirably, it did in McCabe's $V(G)$ formulation of $E - N + 2P$.

This metric is easily evaluated and gives an indication of the number of predicates and selection alternatives that affect concurrency-related control flow. This is analogous to McCabe's observation that the cyclomatic complexity of a program represented by a single strongly connected graph is $\pi + 1$, where π is the number of predicates in the program. C_2 does not depend on overall program size, in terms of number of tasking features. Furthermore, it only depends on the features of tasks in isolation and is not sensitive to the program context of task interactions.

4.4 Information Flow Metric

The information flow model has been successfully used as a basis for metrics for describing the structure of distributed programs [Rom87]. These metrics were used to explain or predict the qualities shown in Table II. Maintainability was the average effort, in staff-

Table II: **Correlation of Complexity Metrics to Maintenance-Related Qualities:** Spearman Correlation Coefficients, all significant at the 0.001 level or better (extracted from [Rom87])

quality	*external (information flow)*	*internal*	*combined*
Maintainability	0.75	0.7	0.85
Comprehensibility	0.8	0.66	0.74
Locality	0.8	0.78	0.83
Modifiability	0.62	0.8	0.81

hours, required per maintenance assignment. Comprehensibility was the average effort required to isolate the program units affected by a modification. Locality was based on the average number of program units that were modified in each maintenance assignment and the average maximum portion of a maintenance assignment restricted to one program unit. Modifiability was based on the average effort required for a correction, after isolation, per maintenance assignment and per program unit.

The external metrics were based on the information flow model. The internal metrics were program length or structure, except for locality, for which the internal metric was the intensity of unit embedding as measured by the number of send/receive operations

and procedure calls. The external metrics were better predictors or explainers of maintainability, comprehensibility, and locality. The internal metrics were noticeably better for modifiability; though not of interest for analyzability, it is important for restructuring. Combined external and internal metrics correlated better than either individually, suggesting that such combinations should be considered in this study as well. We do this by incorporating both kinds of metrics into the metric set.

Our information flow metric is analogous to Henry and Kafura's information flow metric for sequential procedures [HK81, HS90]. Based on the connections between the procedure and its environment, this metric is the square of the product of the fan-in and fan-out of the task. Fan-in is the sum of the flows into the task and the number of global data structures read, while fan-out is the sum of out flows and global data structures written. An analogous TIG-based metric is:

$$C_3 = \left(\sum_{i=1}^{T} (in\text{-}edges_i \cdot out\text{-}edges_i) \right)^{ln\ T}$$

where $in\text{-}edges_i$ is the sum of task entries and shared variables read in task i, and $out\text{-}edges_i$ is the sum of entry calls and shared variables written by task i.

There are two differences between C_3 and the original information flow metric. First, the former is summed over all tasks, while the latter is a separate value for each procedure. The aggregation provides a measure for the entire program. Second, the former is raised to the power $ln\ T$, while the information flow exponent is two based on weights used in Brook's law of programmer interaction and Belady's formula for system partitioning. We elected to replace the constant with a weak function of the number of tasks to reflect the exponential dependence on T. It conveniently allows the metric to vary with the natural logarithm of both the sum of the in-edge, out-edge products and T, because $(sum\ of\ products)^{ln\ T} = T^{ln(sum\ of\ products)}$. At this point, we have no strong basis for giving more weight to either the sum of products or T.

As an example of the interpretation of C_3, a program structure based on a pure client-server architecture has a C_3 value of 0. A task with no out-edges is a server, while a task with no in-edges is a client.

5 Metric Evaluation

The metric evaluation is based on metric *validity*, defined here as the accuracy of the metric's assessment of analyzability expressed in terms of concurrency graph size. TIG-based metrics offer two advantages for rapidly and accurately assessing analyzability. First, they suppress nonessential details, in this case actions not related to concurrency. They are minimal in that any other metric could not better reflect concurrency-related characteristics given the suppression of all activity not related to concurrency. Second, TIGs can be mechanically generated from the program source code and automatically evaluated with the metrics.

Weyuker developed a set of properties for evaluating syntactic software complexity measures [Wey88]. The properties are based on a *program*, consisting of a *PROGRAM statement* which defines all input variables, a *program body*, and an *OUTPUT statement*. A program body may include assignment, conditional, or iterative statements, or sequences of program bodies. For convenience in referring to these nine properties, we summarize and attach a catchphrase to each as shown in Table III. The properties are used to evaluate a metric for evaluating the complexity, denoted by $|P|$, of a program P.

Damerla and Shatz evaluated their γ' metric against each of these criteria. They defined a program body to be a set of tasks that communicate (either directly or indirectly), and because distinct program bodies cannot interact, concluded that Properties 6 (context sensitivity) and 9 (component interaction sensitivity) were not applicable to their measures. However, this definition precluded meaningful interpretation of program body composition, which was explicit in the original definition. Instead, we adopt a program body definition suited to concurrent programs by augmenting the original definition of program body with task entry, entry call, and nondeterministic choice (`select`) statements. The evaluations of our metrics are summarized in Table IV. YES and NO signify whether the criteria does or does not, respectively, satisfy the indicated property.

The evaluation with Properties 1 through 5 and 8 verifies that the metrics meet the noted properties, other than all not being sufficiently fine, but does not distinguish the metrics. W, the TICG size estimator, and C_2, based on cyclomatic complexity, do not

Table III: **Concurrent Program Evaluation Properties:** P, Q, and R refer to arbitrary program bodies and $|P|$ denotes the complexity of P (based on [Wey88])

property number	*catchphrase*	*summary description*
1	inequality	$(\exists P)(\exists Q)(\|P\| \neq \|Q\|)$
2	sufficiently fine	$(\forall c)(c \geq 0)(\exists$ a finite number of $P)(\|P\| = c)$
3	sufficiently coarse	$(\exists P)(\exists Q)(P \not\equiv Q \ \wedge \ \|P\| = \|Q\|)$
4	implementation dependency	$(\exists P)(\exists Q)(P \equiv Q \ \wedge \ \|P\| \neq \|Q\|)$
5	monotonicity	$(\forall P)(\forall Q)(\|P\| \leq \|P;Q\| \ \wedge \ \|Q\| \leq \|P;Q\|)$
6	context sensitivity	$(\exists P)(\exists Q)(\exists R)(\|P\| = \|Q\| \ \wedge \ \|P;R\| \neq \|Q;R\|)$; $(\exists P)(\exists Q)(\exists R)(\|P\| = \|Q\| \ \wedge \ \|R;P\| \neq \|R;Q\|)$
7	statement order responsivity	$(\exists P)(\exists Q)(Q$ is a permutation of $P \wedge$ $\|P\| \neq \|Q\|)$
8	renaming insensitivity	$(\forall P$ that are renamings of $Q)(\|P\| = \|Q\|)$
9	component interaction sensitivity	$(\exists P)(\exists Q)(\|P\| + \|Q\| < \|P;Q\|)$

Table IV: **Metric Evaluation Summary**

property number	*catchphrase*	W	C_1	C_1'	C_2	C_3
1	inequality	YES	YES	YES	YES	YES
2	sufficiently fine	NO	NO	NO	NO	NO
3	sufficiently coarse	YES	YES	YES	YES	YES
4	implementation dependency	YES	YES	YES	YES	YES
5	monotonicity	YES	YES	YES	YES	YES
6	context sensitivity	NO	YES	YES	NO	NO
7	statement order responsivity	NO	NO	YES	NO	YES
8	renaming insensitivity	YES	YES	YES	YES	YES
9	component interaction sensitivity	NO	YES	YES	NO	YES

offer context or component interaction sensitivity or statement order responsivity; this reflects their purely syntactic basis. C_3, based on information flow, reflects the external task structure in terms of the intertask ordering of task interaction statements. C_1 and C_1', which measure nondeterminism, are the only metrics that reflect both context and component interaction sensitivity and therefore consider synchronization and communication between tasks. Property 7 discriminates between C_1 and C_1' for intratask statement reordering only. None of the evaluated metrics are sensitive to intertask statement reordering.

Metrics that satisfy Property 9, component interaction sensitivity, are particularly useful when deciding whether a program structure lends itself to compositional analysis, and when considering how to parcel the analysis. An extreme example program consists of components with no interaction. This is an ideal candidate for compositional analysis and has the highest payoff in terms of state-space reduction: the reachability graph of the entire program includes the cross product of all individual component actions, though there can be no concurrency anomalies. Metrics must satisfy Property 9 in order to identify groups of components that are, for evaluation purposes (or are close to, for guiding restructuring), connected components.

6 Examples

To demonstrate the efficacy of the reengineering-for-analysis approach, highlights of its application to two examples are reviewed. The elevator example was developed starting from formal requirements, through Z and RTIL specifications, and with assistance of a CASE design tool. However, it was not specifically designed to be analyzed. The number of elevators was varied to explore the concurrency state-space size: the state space of the three-elevator version was too large to be analyzed with the CATS concurrency analysis tool suite [YTL+92]. The gas station example is a small but familiar program that was developed as a vehicle for describing techniques for debugging concurrent programs. It is considered here because the metrics point to a worthwhile design change.

6.1 Elevator

The elevator example is a simplified model of an elevator system (Z and RTIL specifications appeared in [RAO92]). The controller includes a single Controller task and one Doorman task for each elevator. An elevator task provides a uniform interface to the various elevators, which are each modeled by a task. Finally, a driver task simulates the arrivals of user requests and calls. A qualitative indication of the complexity of the tasking structure is given by Figure 4 using a simplified variation of Buhr diagrams.

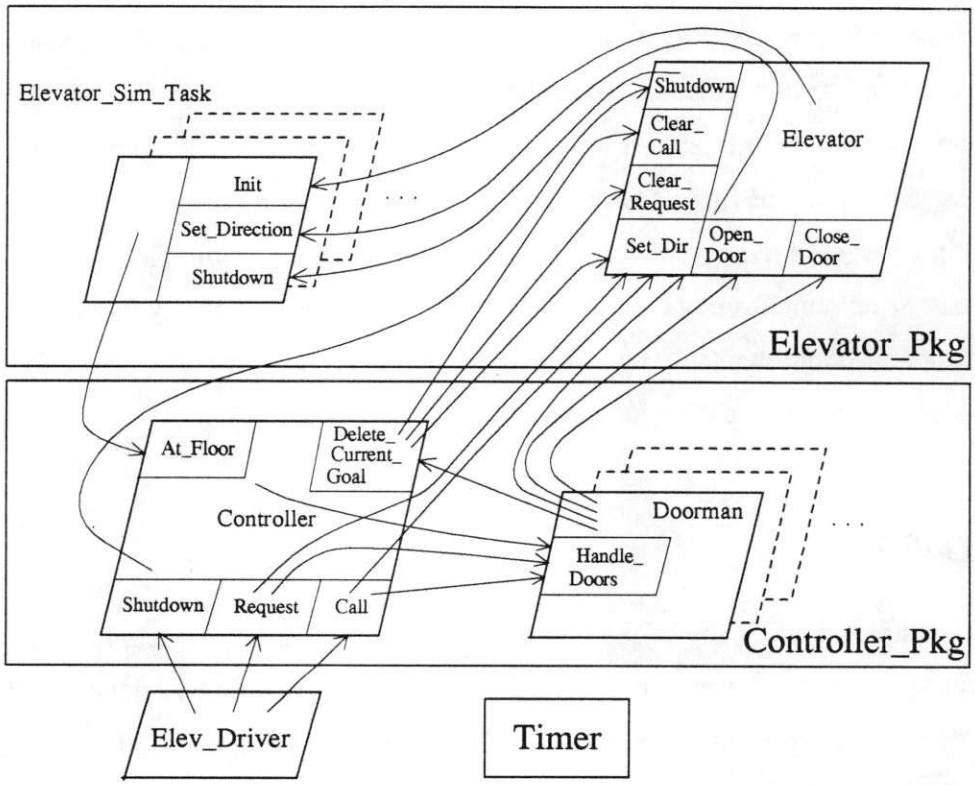

Figure 4: **Elevator Task Structure Diagram**

Rectangles depict packages. Parallelograms represent tasks, with arrows from callers to entries. Shadowed parallelograms with dashed lines indicate optional multiple task type instances.

Our original goal was to construct and analyze the concurrency graph of a three-

Table V: **Metric Scores for Two Versions of Elevator Example**

metric	1 elevator		2 elevators		3 elevators
	original	*restructured*	*original*	*restructured*	*restructured*
TICG states	1933	463	121727	6268	67415
W	2032	760	33215	3605	16902
CC	35	33	50	43	53
γ	16	16	26	22	28
γ'	32	9	299	55	277
C_1	19	21	33	28	35
C_1'	35	9	314	56	281
C_2	92	70	108	95	122
C_3	1723	997	13824	3562	10935

elevator example. However, TICG sizes (and build times[4]) for one- and two- elevator examples of 1933 (15 seconds) and 121,727 (6882 seconds), respectively, indicated that this was not attainable. Restructuring paid off, as shown by the metrics in Table V: the three elevator TICG had 67,415 states and took 752 CPU seconds to build.

The component of C_3 contributed by each task is shown in Figure 5. The "edges product" column shows the product of in-edges, or task entries, and out-edges, or entry calls, for each task (there are no shared variables in this example). While the Controller and Elevator tasks were clearly more complex than the others, the Doorman and Elevator_Sim_Task tasks were replicated for each elevator (signified by the $\_n$ task name suffix), and therefore contributed to both the base and exponent in the formula for calculating C_3. This strongly suggested trying to eliminate some of these tasks.

Inspection of the task structure diagram and source code revealed that the Doorman tasks were not necessary. The primary purpose of the Doorman tasks was to permit the controller task to service other elevators while the doors of one elevator were held open for a minimum duration. Such tasks did not correspond directly to features of the problem. Design methodologies based on task decomposition, such as entity-life modeling [San89], would not have produced designs with these tasks. Therefore, a restructuring was performed, with the open-door delay placed in the Elevator_Sim_Task modelling each

[4]The CPU times were obtained from the Unix `time` command (user + system), on a 64 Mb Sun 4/670, not under controlled conditions though on a lightly loaded system.

```
TIG metric evaluator, version 0.8

 TIG   TIG     in-  out-    edges    task
nodes edges   edges edges  product   name
----- -----   ----- -----  -------   ----
   35    79       5    12       60   controller
    5     8       0     2        0   delayexpire
   11    11       1     4        4   doorman_1
   11    11       1     4        4   doorman_2
    7     8       0     3        0   elev_driver
   21    56       6     4       24   elevator
    9    17       3     1        3   elevator_sim_task_1
    9    17       3     1        3   elevator_sim_task_2
----- -----   ----- -----  -------
  108   207      19    31       98   (8 tasks)

121727 TICG states
W = (108/8)^(8/2) = 33215

CC = 19 + 31 = 50
alpha  =   5; beta  =  21; gamma  =   5 +  21 =  26
alpha' =   1; beta' = 298; gamma' =   1 + 298 = 299

C1  = 12 + 21 = 33
C1' = 16 + 298 = 314
C2  = 207 - 108 + 8 + 1 = 108
C3  = 98^ln(8) = 13824
```

Figure 5: **Elevator Task and Program Metrics:** original design, two elevator configuration

elevator. The TICGs are considerably smaller, and the three-elevator example can now be analyzed.

The Damerla and Shatz γ' metric decreased considerably with restructuring, as shown in Table V. This was primarily due to a reduction in β', in turn due to removal of Doorman tasks with their `select` statements and calls on entries in `select` statements of the Elevator and Controller. C_1', with its origin in this metric, showed the same trend. C_2 dropped primarily because of the reduction in the number of TIG edges with the removal of the Doorman task and the associated connections in the Controller task. C_3 decreased solely because the numbers of tasks decreased; the edge-product sum increased with restructuring.

6.2 Gas Station

The gas station example was introduced by Helmbold and Luckham [HL84]. Though the TICGs are (for small, "standard" problem versions) small enough to be easily managed, analysis with the metric set pointed to an interesting potential design modification.

Ada tasks model multiple gasoline pumps and customers and a single operator. A customer prepays the operator who then activates the requested pump. After pumping, the customer is given change by the operator. The metric scores for the two customer, one pump configuration are shown in Table VI.

Table VI: **Metric Scores for Two Versions of Gas Station Example**

metric	*original*	*reversed Change entry*
TICG states	118	117
W	100	90
CC	18	17
γ	5	6
γ'	6	7
C_1	6	7
C_1'	7	8
C_2	18	11
C_3	51	21

Inspection of the original version scores revealed that reversing the direction of the Change entry call, from Operator-to-Customer to Customer-to-Operator, made Operator a server task (with no out-edges). In addition, the reversed entry call is a better model of the real problem: a customer issues a request for change to the operator, not the other way around.[5]

The TICG size dropped by 1 state out of 118 with this structural modification. The

[5] As noted by Helmbold and Luckham, deadlock freedom for our original version (Figure 4 in [HL84]) requires a runtime scheduler with FIFO run queue and no preemptive timeslicing. Deadlock is possible for the case where a customer prepays and rushes to the pump, reaching it before a customer that had prepaid earlier. In the reversed Change entry version, there will not be deadlock in this situation; rather, the operator gives the second customer change based on the prepayment amount of the first customer. For both versions, a possible solution to this deficiency is to allow a pump to be used only by the customer that has prepayed on it.

Damerla and Shatz γ and γ' metrics increased very slightly, as shown in Table VI. This was due to slight increases in α and α', respectively, because Customer callers could then queue up on the single Change entry in the Operator task[6]. In the original configuration, only the single Operator task could call any Customer's Change entry. Similarly, C_1 and C_1' increased. The metrics most sensitive to this restructuring were C_2, due to the reduced number of TIG edges in the Customer and Operator tasks, and C_3, due to the removal of all out-edges from the Operator task. It then behaved as a server task and did not contribute to C_3.

7 Related Work

There has been considerable progress in developing structural complexity metrics and program restructuring approaches for sequential programs, and to a much lesser extent, concurrent programs. In addition to briefly reviewing these topics, we consider an analysis augmentation to concurrent program design methodologies.

Structural complexity metrics for sequential programs have been available for some time. Harrison, *et al.*, surveyed and evaluated program size, control flow, and data flow metrics [HMKD82]. Côté, *et al.*, comprehensively surveyed metrics, including structural metrics, emphasizing those that have appeared since 1980 [CBOR88]. In contrast, metrics for evaluating the tasking structure of concurrent programs are limited to those of Wampler and Damerla and Shatz, as described in Section 4.

Program restructuring has been investigated for a variety of purposes. Two of these related to reengineering for concurrency analysis are restructuring of sequential programs to execute concurrently, and restructuring during maintenance. Larus developed CURARE, a restructurer that transforms sequential Lisp programs into concurrent programs for execution on shared-memory multiprocessors [Lar89]. (Larus notes that parallelization is a more descriptive though less appealing term than restructuring.) The primary focus is on parallelization of recursive operations, which are treated as loops. CURARE operates automatically, though declarations may be added to the source code to reduce the conser-

[6] And potentially receive the wrong change, as noted above.

vative extent of static analysis.

Restructuring during maintenance is performed in order to enhance program understandability. Griswold and Notkin developed an automated restructuring approach that includes semantics-preserving transformations [GN92]. In contrast, reengineering-for-analysis transformations do not, in general, preserve meaning within the design and implementation abstraction levels.

Avrunin, *et al.*, propose augmenting conventional concurrent program design processes with static analysis by incorporating the following stages [ADWR86]:

- design formulation
- constrained expression generation
- analysis

Design formulation proceeds with tools and notations familiar to the designer. The constrained expressions are generated mechanically. Static analysis results confirm intended system behavior or reveal areas that require modification or further development. The weakness with this approach is the lack of a mechanism for assessing the analyzability of the design; as with other static concurrency analysis methods, no test exists for analyzability other than attempting to analyze a model of the program. Though this design process minimizes designer effort on the first design formulation, the tradeoff is that analysis provides minimal useful feedback if the program is too large to be analyzed, and no guidance for restructuring to enable analysis.

8 Conclusion

The key to metric-driven reengineering is the validity and the comprehensiveness of the metrics. The proposed metric set is intented to be a firm starting point; metrics will be added and refined as experience accrues. The evolution process of Figure 2 provides a framework for guiding metric set development.

An important direction for further study is automated identification of analysis parcels. Efficiently and effectively isolating groups of tasks that can be analyzed apart from the remainder of the program is necessary for analyzing large-scale programs. A metric-driven

approach should be well suited to identifying groups of tasks with minimum interaction with tasks in other groups.

The focus to date has been on reengineering of existing programs. A natural extension is to apply the lessons learned about the effect of structure on analyzability earlier in the life cycle. In particular, a long-term goal is to derive concurrent program design rules that assure analyzability. Ultimately, these could be incorporated into concurrent program design methodologies to address analyzability earlier in the life cycle.

References

[ADWR86] George S. Avrunin, Laura K. Dillon, Jack C. Wileden, and William E. Riddle. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Transactions on Software Engineering*, SE-12(2):278–292, February 1986.

[CBOR88] V. Côté, P. Bourque, S. Oligny, and N. Rivard. Software metrics: An overview of recent results. *Journal of Systems and Software*, 8(2):121–131, March 1988.

[CC90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–18, January 1990.

[DS92] Srinivasarao Damerla and Sol M. Shatz. Software complexity and Ada rendezvous: Metrics based on nondeterminism. *Journal of Systems and Software*, 17(2):119–127, February 1992.

[GN92] William G. Griswold and David Notkin. Automated assistance for program restructuring. Technical Report CS92-221, University of California, San Diego, January 1992.

[HK81] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, September 1981.

[HL84] David P. Helmbold and David C. Luckham. Debugging Ada tasking programs. In *Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments*, pages 96–105, St. Paul, October 1984.

[HMKD82] Warren Harrison, Kenneth Magel, Raymond Kluczny, and Arlan DeKock. Applying software complexity metrics to program maintenance. *IEEE Computer*, 15(9):65–79, September 1982. Also appeared in the IEEE Computer Society *Tutorial on Software Restructuring*, order number 680.

[HS90] Sallie Henry and Calvin Selig. Predicting source-code complexity at the design stage. *IEEE Software*, 7(2):36–45, March 1990.

[Lar89] James R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California, Berkeley, March 1989.

[LC89] Douglas L. Long and Lori A. Clarke. Task interaction graphs for concurrency analysis. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 44–52, Pittsburgh, May 1989.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[Mur91] James A. Murdock. *Perturbations: Theory and Methods*. John Wiley & Sons, New York, 1991.

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the Fourteenth International Conference on Software Engineering*, Melbourne, Australia, May 1992.

[Rom87] H. Dieter Rombach. A controlled experiment on the impact of software structure on maintainability. *IEEE Transactions on Software Engineering*, SE-13(3):344–354, March 1987.

[San89] Bo Sanden. An entity-life modeling approach to the design of concurrent software. *Communications of the ACM*, 32(3):330–343, March 1989.

[Sel92] B. Henderson Sellers. Modularization and McCabe's cyclomatic complexity. *Communications of the ACM*, 35(12):17–19, December 1992.

[Sha88] Sol M. Shatz. Towards complexity metrics for Ada tasking. *IEEE Transactions on Software Engineering*, 14(8):1122–1127, August 1988.

[TLK92] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, March 1992.

[Wam85] Gordon Kent Wampler. Static concurrency analysis of Ada programs. Master's thesis, University of California, Irvine, 1985.

[Wey88] Elaine J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, September 1988.

[YTL+92] Michal Young, Richard N. Taylor, David L. Levine, Kari Forester, and Debra Brodbeck. A concurrency analysis tool suite: Rationale, design, and preliminary experience. Technical Report TR-128-P, Software Engineering Research Center, 1398 Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, October 1992.

[YY91] Wei Jen Yeh and Michal Young. Compositional reachability analysis using process algebra. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification (TAV4)*, pages 49–59, Victoria, British Columbia, October 1991. ACM SIGSOFT, ACM Press.