AFRL-IF-RS-TR-2005-133
Final Technical Report
April 2005

PACEMAKER: CONTINUOUS VALIDATION OF COMPLEX SYSTEMS

University of Oregon

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K512

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-133 has been reviewed and is approved for publication

APPROVED: /s/
DEBORAH A. CERINO
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Advanced Computing Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 2005	**Final** **Jul 00 – Oct 03**

4. TITLE AND SUBTITLE

PACEMAKER: CONTINUOUS VALIDATION OF COMPLEX SYSTEMS

5. FUNDING NUMBERS

G - F30602-00-2-0620
PE - 62302E
PR - DASA
TA - 00
WU - 01

6. AUTHOR(S)

Michal Young
Stephen Fickas

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Oregon
Department of Computer and Information Science
Eugene OR 97403-1202

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/IFT
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2005-133

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Deborah A. Cerino/IFT/(315) 330-1445 Deborah.Cerino@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	

13. ABSTRACT ***(Maximum 200 Words)***

It is currently difficult to develop software systems that are dynamically composable/re-composable, i.e., systems that can be dynamically assembled/adapted at run-time based on feedback from software monitors (gauges) that provide information on both functional and non-functional system properties. The goal of the University of Oregon research was to develop technology that will enable mission critical systems to meet these high assurance, high dependability, and high adaptability requirements. In particular, this effort generalized simple 0/1 gauges such that these gauges can now have a range of values including a “yellow zone” that indicates an impending problem before the problem becomes critical.

14. SUBJECT TERMS

Architecture-based adaptation, logical models of software

15. NUMBER OF PAGES 39

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

Overview..1

Main Results 1

Conclusions..2

Refactoring Design Models for InductiveVerification……………………………………3

Towards Scalable Compositional Analysis by Refactoring Design Models 8

Flow Equations as a Generic Programming Tool for Manipulation of Attributed Graphs………………………………………………………………………………….18

Refining Code-Design Mapping with Flow Analysis…………………………………….26

1. Overview

The goal of University of Oregon research under the Dynamic Assembly for Systems Adaptability, Dependability, and Assurance (DASADA) program was to provide (in collaboration with other DASADA projects) facilities that generalize simple 0/1 gauges (has something bad happened?) to gauges with a range of values including, at a minimum, a “yellow zone” that indicates an impending problem before it has become too late to avoid it. To reach this goal, we pursued two threads of development. First, we investigated techniques to derive from off-the-shelf model-checking tools an abstract model in the form of an automaton whose states are labeled by gauge regions. Second, to associate transitions in this abstract model with events in an actual implementation, we investigated techniques for fusing and transforming information derived from architectural design models and implementations. As the project evolved, additional emphasis was placed on flexible tool support for extracting and refining architectural models.

2. Main Results

A simple facility for monitoring a “yellow-zone” gauge was produced and demonstrated early in the project. Much of the remainder of the project effort was concerned with obtaining useful models (structural and behavioral) and maintaining or monitoring its correspondence with a design-level model.

One thread of this research involved “refactoring” design models for verification. The motivation for this is that a model that follows the implementation structure of a complex system is seldom verifiable (using, for example, modern model-checking tools). A model that is cleanly organized around a logical system structure is better for both informal reasoning and verification, but it is difficult to ascertain its correspondence with the system “as-built.” We posited that this could be overcome by verifying a set of sound transformation steps between two different models, one a logical architecture and the other an “as-built” model of the implementation. This is analogous to “refactoring” object oriented designs, but whereas the standard refactorings are simple structural transformations of code (e.g., encapsulating a field of an object), this more ambitious “refactoring” involves transformation of a detailed behavioral model in tandem with structural transformations. The logical architecture can be subjected to model checking, while the “as-built” model is the source of implementation conformance checks. We demonstrated the feasibility of this approach, described in

Refactoring Design Models for Inductive Verification. Yung-Pin Cheng. *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, Roma, Italy, July 2002. Pages 164-168.

Towards Scalable Compositional Analysis by refactoring design models. Yung-Pin Cheng, Michal Young, Che-Ling Huang, and Chia-Yi Pan. *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering,* Helsinki, Finland, June 2003, pages 247-256.

The second main thread of the research was development, refinement, and evaluation of light-weight tool support for combining and manipulating information about system structure and behavior. By "light-weight" we mean that (a) the tool was scriptable and could be used for a variety of ad hoc and canned analyses, quickly modifying and augmenting those analyses for a particular problem and (b) the information manipulated by the tool could be easily extracted from a variety of sources, from execution monitoring or models or source code analysis, from simple ad hoc extraction tools as well as more sophisticated systems.

The GenSet system, begun in earlier DARPA-sponsored research, was significantly extended in the Pacemaker project. In one evaluative experiment, a graduate student used GenSet to validate and transform a C2 architectural model extracted from an Airborne Warning and Control System (AWACS) model, obtained initially from Lockheed-Martin. Two other experiments with GenSet resulted in published papers. The first describes the underpinnings of GenSet and its application to extracting a simple architectural model from the Linux system kernel, demonstrating that this simple scriptable tool could be competitive with special-purpose reverse engineering toolkits:

Flow Equations as a Generic Programming Tool for Manipulation of Attributed Graphs. John Fiskio-Lasseter and Michal Young. *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering,* Charleston, South Carolina, November 2002. Pages 69-72.

In the most recent evaluation (carried out after the end of funding, as an extension of the funded work), GenSet was again applied to a reverse engineering task. Whereas the 2002 paper demonstrated that GenSet could effectively reproduce a standard reverse engineering task, the recent paper proposes a new analysis for which there was no prior tool support, and reports experience using GenSet to carry out that analysis. Since the analysis itself was designed and refined iteratively while using the tool, the ability to devise and try ad hoc analyses using a scriptable tool was crucial.

Refining Code-Design Mapping with Flow Analysis. Xiaofang Zhang, Michal Young and John H. E. F. Lasseter. *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering,* Newport Beach, California, November 2004. Pages 231-240.

3. Conclusions

A major theme of this work has been reconciling formal, logical models of software *as we want to reason about it* with more complex, sometimes imperfect models of software *as it was actually constructed*. In many cases it is not possible for one model to serve both purposes, at least without herculean efforts that are soon obsoleted by further system evolution, but we have shown that a great deal of automation is possible in establishing, monitoring, and refining relations among models. This effort is part of the DARPA Dynamic Assembly for Systems Adaptability, Dependability, and Assurance (DASADA) Program.

Refactoring Design Models for Inductive Verification * †

Yung-Pin Cheng
Dept. of Information and Computer Education
National Taiwan Normal University
162 Sec.1 Hoping E. Rd.
Taipei 106
Taiwan
ypc@ice.ntnu.edu.tw

ABSTRACT

Systems composed of many identical processes can sometimes be verified inductively using a network invariant, but systems whose component processes vary in some systematic way are not amenable to direct application of that method. We describe how variations in behavior can be "factored out" into additional processes, thus enabling induction over the number of processes. The process is semi-automatic: The designer must choose from among a set of idiomatic transformations, but each transformation is applied and checked automatically.

Keywords

Refactoring, Network Invariants, Parameterized System, Compositional Analysis, Concurrency

1. INTRODUCTION

When applying finite-state verification methods to a concurrent system, the system is modeled as several finite-state machines which communicate among themselves or with their environment. A major limitation of the technique is that the number of processes, the number of communication channels, and the number of data values of the model must be concrete. However, in practice, systems may be parameterized by size. Parameterized systems induce infinite state space, which makes finite-state verification methods inapplicable.

To extend finite-state verification methods to a parameterized system with many identical processes, one would prefer to perform an inductive verification that applies to arbitrary size instances of the system. A popular approach to verifying systems parameterized by size is to construct a so-called network invariant and then check if the invariants pass the test of an induction framework (explained in section 2) such as those used by Wolper [9] and Kurshan [5].

The induction framework, however, assumes the behaviors of a process are constant, i.e., the finite-state machines representing the behaviors are fixed – with constant transition relations and number of states. This assumption could be true for some hardware systems, some protocols in which processes communicate by a shared bus, or linearly structured systems in which processes only communicate with its right or left neighbor. However, in many other application domains, one or more of the individual processes varies in some systematic way depending on the size of the system.

Standard induction frameworks cannot be directly applied to systems in which individual process behaviors are parameterized by system size. We describe how models of such systems can be transformed by *refactoring* to make induction applicable. The transformations "factor out" the variations in behavior into additional processes. Each refactoring step maintains equivalence between the original processes and compositions of the factor processes, so that the final refactored model is equivalent (weakly bisimilar) to the original model. Refactoring can be useful for improving the complexity of compositional state-space analysis in general and is particularly useful for enabling inductive analysis.

This paper is organized as follows: We review the inductive finite-state verification in Section 2. In section 3, we describe the refactoring technique and its application to an example system. In section 4, we illustrate how the refactored example system can be verified inductively. Section 5 is a discussion. Section 6 and section 7 end the article with related work and conclusions.

2. INDUCTION

In practice, systems can be parameterized by many ways. They can be parameterized by the number of identical com-

*This effort was supervised by Michal Young, Dept. of Comp. and Info. Sciences, University of Oregon while the author was a Ph.D student at CS, Purdue University. It was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0034. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

†A full version of this paper is available at `http://vulcans.ice.ntnu.edu.tw/ypc`.

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2002 ACM 1-58113-562-9...$5.00

ponents, the number of data length (e.g., the length of a bounded buffer) and data values, or the number of control commands (e.g., a protocol that allows retransmission of messages over a lossy channel at most n times). Let a system S of size i be denoted as S_i and let all S_i form a family $F = \{S_i\}_{i=1}^{\infty}$. Let φ be some property of interest. The problem of verifying a parameterized system is to answer whether every S_i in F satisfies φ.[1]

One popular approach to address the problem is as follows: Consider systems parameterized by the number of identical components. Let I be some identical component and S_0 be the control process. We have for $i \geq 1$, S_i in F, $S_{i+1} = S_i||I$, where '$||$' is some parallel composition operator. Next, we choose an equivalence relation or a preorder relation (see Kurshan [5] and Hennessy [3]) to relate two concurrent systems. Suppose we choose preorder. Let $\preceq$ be a preorder relation over processes, and φ be a property of interest, such that

$$(P \preceq Q) \wedge (Q \models \varphi) \quad \Rightarrow \quad P \models \varphi.$$

If we can find a process Inv, such that (1) $Inv \models \varphi$ (2) $S_1 \preceq Inv$ (3) $Inv \parallel I \preceq Inv$, we can use the fact that parallel composition is monotonic with respect to the preorder to infer from (3) that

$$Inv||I|| \cdots ||I \preceq Inv. \qquad (4)$$

Finally, we infer from (2) and (4) that

$$S_1||I|| \cdots ||I \preceq Inv. \qquad (5)$$

Finally, using $S_{i+1} = S_i||I$ and (1), we conclude that every S_i in F satisfies φ. A process Inv satisfying the above induction step is called *a network invariant*. In general, a network invariant may not exist due to the undecidability.

Parameterization, however, can affect models in two ways:

1. Models add/remove processes when the system size is increased/decreased.

2. The behaviors (transition relation and number of states) of processes grow/shrink when the system size is increased/decreased.

In the literature, systems shown to pass induction framework mostly vary by size in first way. However, there are many systems whose parameterization affects models in second way or both. The induction framework used by Wolper and Kurshan can not be directly applied to systems with parameterized behaviors because $S_{i+1} = S_i||I$ does not hold for these systems, i.e., $S_i \neq S_0||I_1||..||I_i$.

Consider, for example, a system consisting of one control process and many identical slave processes. Let the system be $G_n = S(n) \parallel I_1 \ldots \parallel I_n$, where $S(n)$'s behaviors are parameterized by n and processes I_i communicates with $S(n)$ by a private channel indexed with i. In Fig. 1(a), we show the example's communication structure. Note that if we increase system size by 1, we add I_{n+1} to the system and replace $S(n)$ by $S(n+1)$.

What we can do to fit such a system into the induction framework is "factor" the single process $S(n)$ (as in Fig.

[1]This problem was shown to be undecidable in general [1].

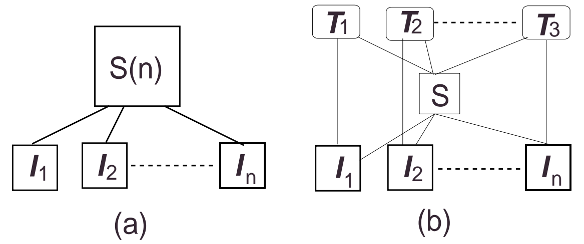


Figure 1: (a) The system G_n before refactoring. (b) The system G_n after refactoring.

1(b)) into a much smaller fixed process S, independent of n, and several identical $T_i, i = 1 \,\text{to}\, n$, with transition labels renamed according to i, where each T_i communicates with the corresponding process I_i. This refactoring can be done in a way that maintains behavioral equivalence, i.e., $S(n)$ is weakly bisimilar to $(S \parallel T_1 \parallel \ldots \parallel T_n)$. So, after refactoring,

$$G_n = (S \parallel T_1 \parallel \ldots \parallel T_n) \parallel I_1 \parallel \ldots \parallel I_n.$$

To increase n by 1, we add a pair of processes $\{T_{n+1}, I_{n+1}\}$ to G_n under the new structure.

Using the new structure of G_n, we can apply the induction framework to verify a system with an arbitrary number of I_i larger than n. If we transform safety properties into a deadlock detection problem (see [2]), then we can use refactoring to inductively verify safety properties.

3. REFACTORING

In this section we present an example to illustrate how the parameterized behaviors of a process can be factored out into individual processes. The refactoring, in fact, is applying a sequence of transformations to a model, where each transformation preserves behavioral equivalence (weak bisimulation) but changes system structure gradually. Note that finding a network invariant is in general undecidable, no one is able to find a fully automated solution that is guaranteed to work for arbitrary systems. So, to refactor a system into a structure that enables inductive verification relies some human wisdom to choose among a set of idiomatic transformations and apply them in a right order. We have constructed a prototype tool to ease the refactoring steps. The tool can display topological view of a system to show the overall communication structure of the model being refactored. If a process in the topological window is clicked, a window displaying the process's state graph will pop up. Then, a user can manually identify parts of the process graph and choose a transformation to apply. If a transformation results in a structure modification, that will be shown in the window of topological view.

3.1 Refactoring a remote temperature sensor system

In this section, a remote temperature sensor system (RTSS) (described originally by Sanden [7] and adapted by Yeh and Young [10]) is refactored and is verified (see section 4) by the induction framework. The remote temperature sensor system is a software-driven system that periodically reports the temperatures of an array of furnace devices. The system is parameterized by the arbitrary number of furnaces. Let the number of furnaces be fn and the furnaces be indexed

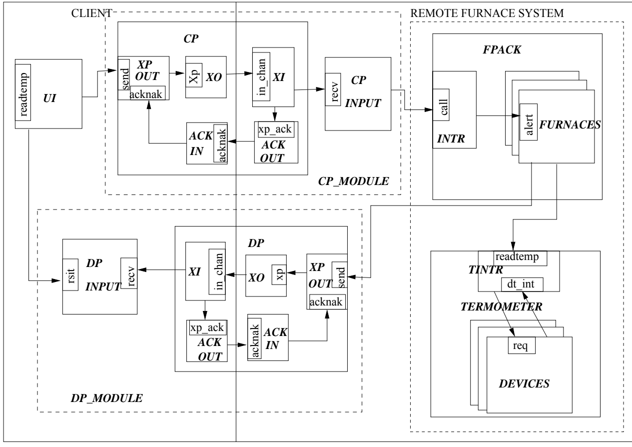


Figure 2: The overall structure of the remote temperature sensor system.

from 0 to $fn-1$. The overall structure of the system is shown in Fig. 2, where component CP and DP implement the alternating bit protocol to deliver packets reliably over a lossy channel. While fn is small, the system can be composed compositionally by a hierarchy (UI (CP CP_INPUT) (FPACK THERMOMETER) (DP DP_INPUT)).

3.1.1 The refactoring of CP_MODULE

We select refactoring CP_MODULE (composed by CP and CP_INPUT) to describe the general idea of refactoring. Fig. 3 shows its behavior with 2 furnaces. To avoid any confusion, please note that in practice the external behavior of a module may not be simple, regular, and manageable. If that is the case, refactoring should be applied to the basic processes such as XP_OUT and XI. In this case, the simplicity of CP_MODULE's external behavior allows refactoring to proceed at a higher level, which can save some effort of refactoring.

In Fig. 3, the process graph inside the box is of CCS semantics. The label name *-send(0)* is a result of symbolic expansion of an Ada[2] statement *accept send(i),* where i=0,1. It should not be understood as a value is passed by the edge. We label an edge with prefix '–' to mean the action is at callee (or server) side. The edge *call_end* models the *do* block of an *accept* statement in Ada. The *accept call(i)* statement in INTR has a *do* block. So, whenever CP_MODULE issues *call(i)*, it must wait for the *do* block in INTR to complete, which is then modeled by *call_end* in CP_MODULE and *-call_end* at the end of *do* block. The box and ports are to illustrate its interfaces to task UI and task INTR.

The overall goal of refactoring is to recognize and separate parts of a process that have essentially been duplicated for dealing with different parts of the system. To accomplish the goal, we need four subsequent transformations below:

Transformation I: Edge relabeling

The first transformation for refactoring CP_MODULE is to help recognition of the variant parts which essentially deal with different furnaces. In CP_MODULE, *-send(0)* and *call(0)* can be easily classified as linked to furnace 0,

[2]Note that our approach does not depend on a particular programming language.

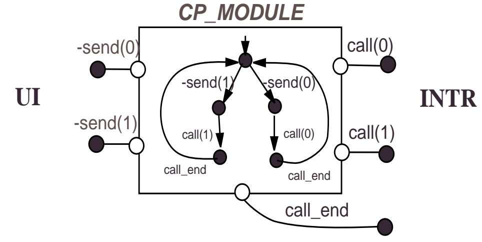


Figure 3: The external behavior of CP_MODULE.

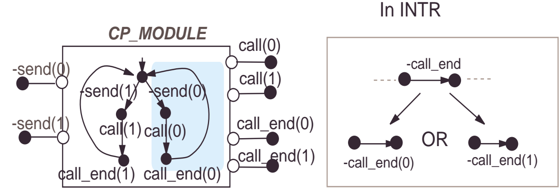


Figure 4: CP_MODULE after edge relabeling transformation. In this transformation, *call_end* is renamed as *call_end(0)* or *call_end(1)* in INTR and CP_MODULE.

-send(1) and *call(1)* can be easily classified as linked to furnace 1, but *call_end* can not. In this example, we intend to classify *call_end* into either linked to furnace 0 or linked to furnace 1 so that CP_MODULE can become a clean, simple task (later shown in Fig. 7) and irrelevant to fn. The intended classification involves renaming *call_end* to either *call_end(0)* or *call_end(1)*. However, such kind of relabeling is not supported by the relabel operator (a.k.a. [a/b]) in CCS. We need a relabeling that is less strict. Recall that *call_end* is an artifact of modeling Ada's *accept/do* semantics. So, a *call_end* is paired to a particular *call(i)*. Using this as guidance, we can relabel CP_MODULE into Fig. 4. Since action names in CCS are in pairs, we need to rename *-call_end* in INTR as well. Consequently, the interface between CP_MODULE and INTR is changed.

To justify the relabeling, we introduce a notion of equivalence. Since CP_MODULE and INTR are modified by this relabeling, CP_MODULE and INTR are viewed as a subsystem. The equivalence we propose is that the subsystem's behavior remains equivalent (weakly bisimilar) before and after the transformation. It can be expressed by the following equation:

$$(CP_MODULE||INTR)\setminus\{call_end\}\approx$$
$$(CP_MODULE||INTR)\setminus\{call_end(0), call_end(1)\},$$

where '≈' is the weak bisimulation of CCS and '\' is the restriction operator of CCS. Verifying the equation can assure the correctness of our relabeling strategy. The general algorithm to determine if a label can be relabeled safely is presented in the full version of this paper.

Transformation II: 1st Behavior decomposition

The second transformation is to decompose CP_MODULE by extracting away the behavior linked to furnace 0. The extracted behavior is then wrapped into a new process. The first step of behavior decomposition is to identify the behav-

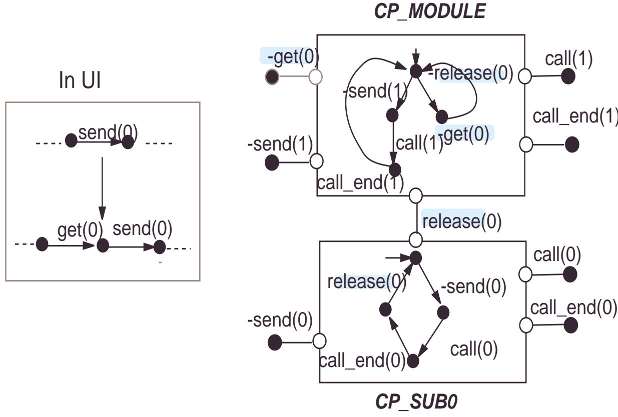


Figure 5: CP_MODULE after Transformation II. CP_SUB0 is the new process created by the transformation. *-send(0)*, *call(0)*, *call_end(0)* are all redirected to CP_SUB0. Every *send(0)* in UI is guarded by new communication *get(0)*.

ior to be extracted. In Fig. 4, we use a shaded area to mark the behavior to be removed. Usually, this is where human assistance is needed. After segments of behavior have been properly identified and marked, the rest of the transformation is automatic. Functions of this transformation include purging the marked behaviors from CP_MODULE, wrapping the marked behaviors into a new task, and inserting new communications to preserve equivalence. Fig. 5 illustrates the result of this transformation. The new task created by this transformation is CP_SUB0 to which *-send(0)* is now redirected. Edge labels highlighted by grey background are the new communications inserted by the transformation. In UI, every edge labeled *send(0)* is replaced by two edges, labeled as *get(0)* and *send(0)*. The equivalence is preserved by:

$$(UI||CP_MODULE) \setminus \{send(0), send(1)\} \approx$$

$$(UI||CP_MODULE||CP_SUB0)\setminus\{send(0), send(1), get(0), release(0)\}.$$

Transformation III: 2nd behavior decomposition

The third transformation is the same as transformation II, only the marked behavior is those linked to furnace 1. The result of this transformation is shown in Fig. 6, where CP_MODULE behaves as a pure semaphore with value 1. For general algorithm which can handle complicated cases in general, please refer to the full version of this paper.

Transformation IV: Semaphore simpli cation

Despite the simple behavior, CP_MODULE in Fig. 6 is still parameterized by fn. To make it independent of fn, we need the forth transformation. But before that, let's review CCS's rendezvous semantics. In CCS, if there are two processes both ready to communicate by action a but there is only one process ready to communicate by co-action $\bar{a}$, the first two processes compete for the rendezvous. This is known as two-way rendezvous, as opposed to multi-way rendezvous of CSP.[3] This important characteristic can be

[3]If CSP is used, this is where CSP models cannot be further simplified to be independent of fn.

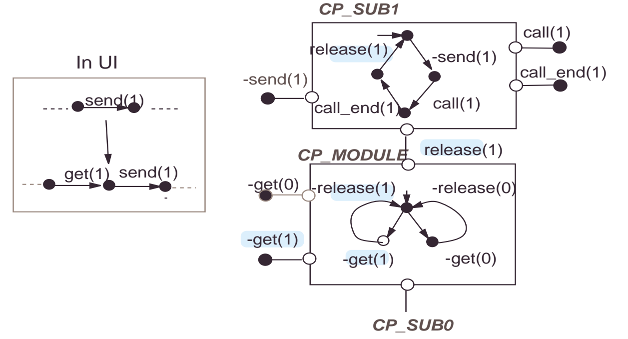


Figure 6: CP_MODULE after Transformation III.

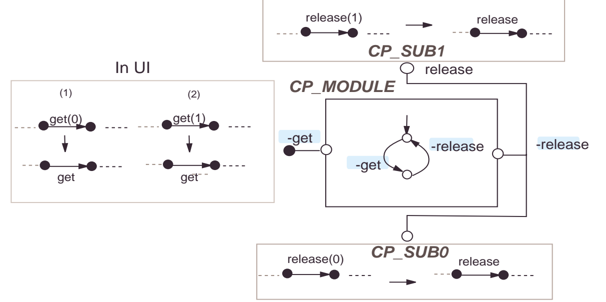


Figure 7: CP_MODULE after Transformation IV, where *get(0)* and *get(1)* are merged into one *get*. So are *release(0)* and *release(1)*.

used to simplify CP_MODULE into Fig. 7, where *get(i)* and *release(i)* are all renamed to new names *get* and *release* respectively. CP_MODULE now only have two ports but each port may have more than one process attached. For example, port *release* is now attached by CP_SUB0 and CP_SUB1. Using the same notion of equivalence as before, the following equation can be verified to justify this simplification:

$$(UI||CP_MODULE)\setminus\{get(0), get(1), release(0), release(1)\} \approx$$

$$(UI||CP_MODULE) \setminus \{get, release\}$$

The nal structure of CP_MODULE

At last, CP_MODULE in Fig. 7 holds a structure amenable to the induction framework. First, CP_MODULE is independent of fn; that is, its process graph no longer varies by fn. Second, this structure is meant to be extended easily: While fn is increased, we simply add another identical process CP_SUB2 and attach its *release* port to that of CP_MODULE.

4. THE INDUCTION OF RTSS

The tasks of refactored RTSS are summarized in Table 1. In second column, task names in bold-italic font are semaphore tasks which are independent of fn. Let

$$C = CP_MODULE||INTR||TINTR||DP_MODULE||UI,$$

Task name	Names of tasks after refactoring
CP_MODULE	***CP_MODULE***,CP_SUB0, CP_SUB1
INTR	***INTR***,INTR_SUB0, INTR_SUB1
TINTR	***TINTR***,TINTR_SUB0,TINTR_SUB1
DP_MODULE	***DP_MODULE***,DP_SUB0,DP_SUB1
UI	***UI***,UI_SUB0, UI_SUB1
Device[i]	(not refactored)
Furnace[i]	(not refactored)

Table 1: The task names of refactored RTSS.

$F_i = CP_SUBi||INTR_SUBi||TINTR_SUBi||DP_SUBi||UI_SUBi$, and $R_i = Device[i]||Furnace[i]$. Let $RTSS(1)$ be the system with 1 furnace (starting from index 0). We have $RTSS(1) = C||F_0||R_0$.

Verification of a safety property is translated into a deadlock detection problem [2]. The places to embed inductive safety properties (which holds for arbitrary number of furnaces) are F_i and R_i for all i. If no safety property is embedded, the verification problem is equal to check if $RTSS$ is free of deadlock with respect to arbitrary number of furnaces. Let $RTSS^*(1)$ denote the $RTSS(1)$ embedded with safety properties. Let Inv be $RTSS^*(1)$ with *-get* and *-release* of each semaphore tasks being exported. The induction step is to verify $RTSS^*(1)$ is deadlock free and $(Inv||F_i||R_i) \approx Inv$. If the equation holds, we conclude the safety property is satisfied with respect to arbitrary number of furnaces. Our experiment shows that Inv is an effective network invariant for $RTSS$.

5. DISCUSSION

Besides the remote temperature sensor system, some systems in the literature are also refactored to see if they can pass the inductive verification. The elevator system is another example worth mentioning. Most of the behaviors of elevator system can be refactored as expected but some behaviors can not. They are for task initialization and termination. Their patterns are a sequence of commands which are issued to elevators one after another. We have not yet managed to refactor the pattern as the composition of many identical processes. We choose to ignore the problematic behaviors (since they are only a minor part of elevator's behaviors) and focus on the continuously running parts. Thus, a network invariant can be constructed and the induction framework becomes applicable.

6. RELATED WORK

In invariant-based approaches, finding the network invariant often requires human ingenuity and trial-and-error. Nevertheless, under some particular topology and conditions, automatic computation for the network invariants is possible. An attempt to generalize automatic computation of network invariants is made by Clarke et al [6], which uses context-free network grammar to describe the topology of networks and provide automatic construction of network invariant but the procedure is not guarantee to terminate.

A case study by Valmari and Kokkarinen [8] on lossy channels is, to our knowledge, the approach closest to that described here. Valmari and Kokkarinen studied a protocol with lossy channels in which the system allows messages to be retransmitted at most n times – a channel parameterized by n. The channel's behaviors resemble the initialization and shutdown sequences of the elevator example. They replace the single parameterized channel by the composition of n smaller processes called *counter cells* under CSP (multi-way rendezvous) semantics [4], which captures the spirit of our refactoring but is rather specialized to the particular case they considered.

7. CONCLUSIONS

Inductive verification using network invariants cannot be directly applied to systems in which individual component processes vary in some systematic way depending on the size of the system. We have described how models of such systems can be transformed — *refactored* — into equivalent models in which inductive verification can be applied.

Refactored models are composed in a modular and hierarchical manner to avoid state explosion during an inductive verification. Refactoring, and verification of the soundness of transformation steps, is performed locally so that its cost is not proportional to the size of the system.

8. REFERENCES

[1] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, pages 207–309, 1986.

[2] S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 8:49–78, January 1999.

[3] M. Hennessy. *Algebraic Theory of Processes*. MIT Press Series in the Foundations of Computing. The MIT Press, Cambridge, Massachusetts, 1988.

[4] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[5] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.

[6] E. M.Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. In *ACM Transactions on Programming Languages and Systems*, volume 19, pages 726–750, 1997.

[7] B. Sanden. Entity-life modeling and structured analysis in real-time software design–a comparison. *Communications of the ACM*, 32(12):1458–1466, December 1989.

[8] A. Valmari and I. Kokkarinen. Unbounded verification results by finite-state compositional techniques: 10^{any} states and beyond. In *International Conference on Application of Concurrency to System Design, Proceedings*, pages 75–85, Aizu-Wakamatsu, Fukushima, Japan, March 1998.

[9] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems, Volumne 407 Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, June 1989.

[10] W. J. Yeh and M. Young. Re-designing tasking structure of ada programs for analysis:a case study. *Software Testing, Verification, and Reliability*, 4:223–253, 1994.

Towards Scalable Compositional Analysis by Refactoring Design Models *

Yung-Pin Cheng
Dept. of Information and Computer Education
National Taiwan Normal Univ.
Taipei, TAIWAN
ypc@ice.ntnu.edu.tw

Michal Young
Dept. of Computer and Information Science
University of Oregon
Oregon, USA
michal@cs.uoregon.edu

Che-Ling Huang
Chia-Yi Pan
Dept. of Information and Computer Education
National Taiwan Normal Univ.
Taipei, TAIWAN
{yklm,pan}@ice.ntnu.edu.tw

ABSTRACT

Automated finite-state verification techniques have matured considerably in the past several years, but state-space explosion remains an obstacle to their use. Theoretical lower bounds on complexity imply that all of the techniques that have been developed to avoid or mitigate state-space explosion depend on models that are "well-formed" in some way, and will usually fail for other models. This further implies that, when analysis is applied to models derived from designs or implementations of actual software systems, a model of the system "as built" is unlikely to be suitable for automated analysis. In particular, compositional, hierarchical analysis (where state-space explosion is avoided by simplifying models of subsystems at several levels of abstraction) depend on the modular structure of the model to be analyzed. We describe how as-built finite-state models can be *refactored* for compositional state-space analysis, applying a series of transformations to produce an equivalent model whose structure exhibits suitable modularity. The process is supported by a parser which can parse a subset of Promela syntax and transform Promela code into refactored state graphs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification - *formal methods, model checking.*

General Terms

Algorithms, Design, Theory, Verification.

*This work is partially supported by National Science Council, TAIWAN under GRANT NO. 90-2213-E-003-009. This work has also been supported by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0034.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.
Copyright 2003 ACM 1-58113-743-5/03/0009 ...$5.00.

Keywords

Refactoring, Compositional Analysis, Promela, CCS

1. INTRODUCTION

Although automated finite-state verification techniques and tools have matured considerably in the past several years, they are still fundamentally limited by the well-known state space explosion problem. A variety of techniques have been developed to mitigate space-space explosion. Nonetheless, approaches to increasing the size of system that can be accommodated in a single analysis step must eventually be combined with effective compositional techniques[14, 3, 7] that divide a large system into smaller subsystems, analyze each subsystem, and combine the results of these analyses to verify the full system.

In practice, compositional techniques are inapplicable to many systems (particularly large and complex ones) because their as-built structures may not be suitable for compositional analysis. A structure suitable for compositional analysis must contain loosely coupled components so that every component can be replaced by a simple interface process in the incremental analysis. Moreover, composing the processes and deriving the interface process must be tractable. Otherwise, we need to recursively divide the component into smaller loosely coupled components until every subsystem in the composition hierarchy can be analyzed. However, an ideal structure seldom exists in practice. Designers often structure their systems to meet other requirements with higher priority. It is impractical to ask designers to structure a design in the beginning for the purpose of verifying correctness.

If it is difficult to prove the correctness of a program as originally designed, one may need to prove the correctness of a transformed, equivalent version of the program. This is a notion known as program transformation, which has been widely studied in the area of functional and logic languages. Here, we apply the idea to transform finite-state models to aid automated finite-state verification. In general, the purpose of our transformations is for obtaining, starting from a model P, a semantically equivalent one, which is "more amenable to compositional analysis" than P. It consists in building a sequence of equivalent models, each obtained by the preceding ones by means of the application of a rule. The rules restructure as-built structures which are not suitable for compositional techniques. The goal is to obtain a transformed model whose structure contains loosely coupled components, where processes in

each component can be composed without excessive state explosion. We refer to the process as *refactoring.*

The general approach to refactoring and some refactoring transformations were rst described in [2] conceptually with an example (without explicit algorithms of transformations). That work describes the application of refactoring to construct network invariants for systems with parameterized behaviors, where those systems are originally inapplicable to inductive veri cation. However, the transformations described in [2] were derived on an ad hoc basis. They are unlikely to be automated and applicable for general systems. Here we propose a uni ed approach to accommodate previous ad hoc transformations, extend refactoring to larger class of systems, provide automated tool support, and focus on the major application of refactoring – *compositional analysis*. We report upon a case study involving the Chiron user interface system, comparing analysis performance with results previously reported by Young et al. [16] and Avrunin et al [1].

In past decades, many approaches have been proposed to address the state explosion problem, such as minimizing overall state space, enumerating states implicitly, or abstracting and compacting models. Unlike those approaches which seek improvement in the fundamental techniques, our approach aims for avoiding state explosion at the level of system structure in a compositional fashion.

This paper is organized as follows. In Section 2, we describe the relation between architecture and composition analysis. In Section 3, we give an overview of refactoring. In Section 4, we introduce the refactoring transformations with simple examples. In Section 5, our tool support for compositional techniques is described. In Section 6, we show the results of applying refactoring to two examples. Finally, we end the paper with related work and conclusions.

2. SYSTEM ARCHITECTURE AND COMPOSITIONAL ANALYSIS

When applying compositional techniques to a system, we must divide a system into several subsystems where these subsystems form a hierarchy. Using that hierarchy, we compose processes in a subsystem and replace it by a simpler process which represents the external behaviors of the subsystem (often called an *interface process*[1]). This process works from the bottom of the hierarchy to the top, until whole system is analyzed. Ideally, state explosion can be avoided in this divide and conquer manner but in practice, compositional analysis often yields no savings in analysis effort.

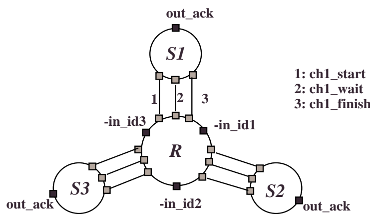


Figure 1: The communication structure of an example system.

Consider an example subsystem in Fig. 1 which consists of processes R, $S1$, $S2$, and $S3$, where $S1$, $S2$, and $S3$ have identical behaviors and R is parameterized by the number of S. In Fig. 2, we show Promela [8] code and state graphs of process R and $S1$. The state graphs are of CCS semantics[12], in which processes communicate by two-way rendezvous. Note that in CCS, paired communications are denoted by a and $\bar{a}$, but we use a and $-a$ instead. In the example, process R iteratively reads an id from channel in (where id is sent by some process which is not in the subsystem) and then uses id to do a sequence of synchronizations with process Si indexed by id. Process Si, after activated by R, tries to send a message ack via a lossy channel out and then return. The ack message is sent to some process which is not in this subsystem. The internal action τ in Si's CCS state graph is to emulate losing message.

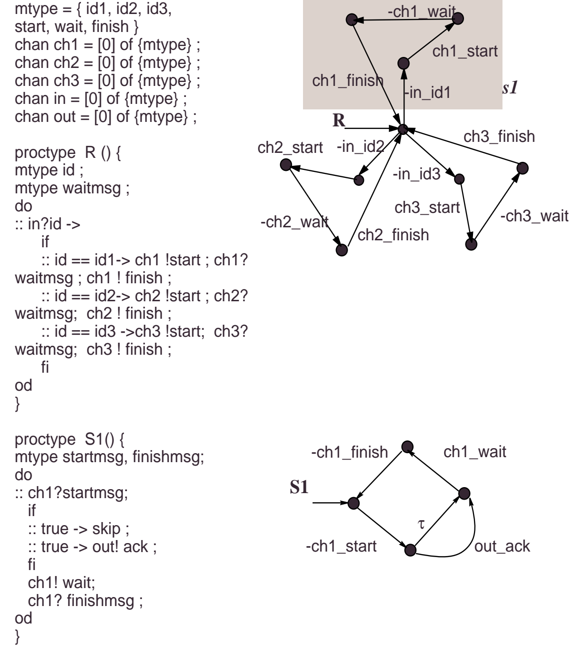


Figure 2: Example process R and S1.

Suppose we want to compose (*R*|*S1*|*S2*|*S3*) in one step. Let *a*={-*in_id1, -in_id2, -in_id3, out_ack*} be the set of ports we must export[2] in the composition. The number of states and transitions generated by parallel composition of (*R*|*S1*|*S2*|*S3*) is 13 states and 18 transitions (see Table 1). After minimized by weak bisimulation, the size becomes 3 states/5 transitions.[3] For larger systems, parallel composition with many processes in one step may suffer state explosion. So, we may try to divide the system and analyze it compositionally. Here, there are few choices to divide the example system. We show three possible subsystems in Table 1, where *b* = {*ch2_start, -ch2_wait, ch2_ ni sh*} and *c* = {*ch3_start,-ch3_wait, ch3_ ni sh*}. Unfortunately, all the subsystems produce state space nearly large as or larger than (*R*|*S1*|*S2*|*S3*). Furthermore, minimizations such as weak bisimulation are much less effective on the state space of these subsystems. Compositional techniques have no merit in this case and sometimes they can even produce worse results [7]. This explains why compositional analysis is thought as a promising approach for combating state explosion but has not yet been widely adopted. In a structure like Fig. 1, no effective subsystems or composition hierarchy can be drawn.

[1]The interface process can be automatically computed from minimizing or abstracting the subsystem state space.

[2]The meaning of exporting ports and restriction operation in CCS are contrary to each other. If a port is exported, it is not restricted in CCS, and vice versa.

[3]The results in Table 1 are computed by Fc2tool[11]

.

Table 1: The state space sizes for different subsystems

	exported ports	*states/ trans*	*minimized*
(R | S1 | S2 | S3)	a	13/18	3/5
(R | S1)	$a \cup b \cup c$	11/14	8/10
(R | S1 | S2)	$a \cup c$	12/16	6/9
(S2| S3)	$b \cup c$	16/40	16/40
(S1|RS1)	d	6/7	3/4

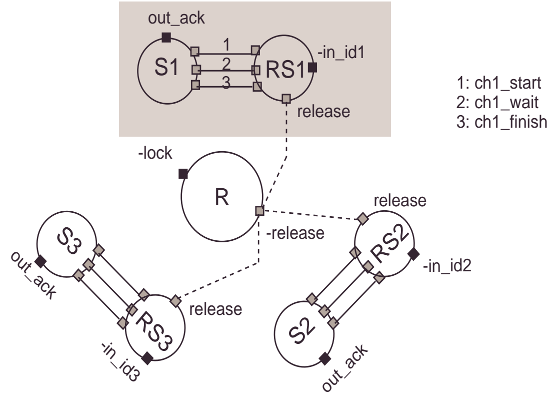


Figure 3: The new structure of refactored example system.

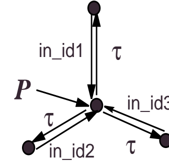


Figure 4: Process P which iteratively invokes *in_idi,i=1 to 3.*

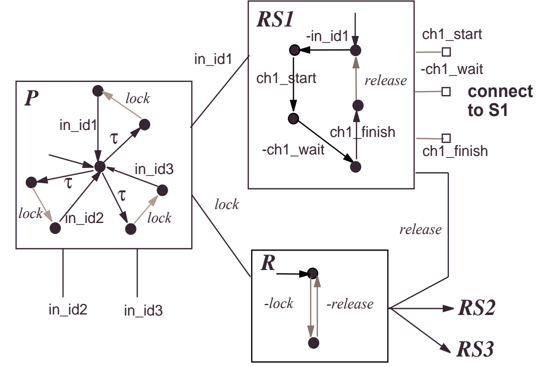


Figure 5: The refactored R,P and the new process RS1.

We say a subsystem is *loosely coupled* to its environment if its interface process contains simple and small state space. So, in a tractable hierarchy, every subsystem must possess such property. Unfortunately, the as-built architectures of many systems do not have this property. In this paper, we propose an approach called *refactoring* to transform a system from an architecture to another with equivalent behavior. For example, refactoring can transform the architecture in Fig. 1 into Fig. 3 by decomposing *R* into *RS1,RS2,* and *RS3*. The behaviors highlighted by shaded region in Fig. 2 are wrapped into a new process *RS1* and the rest is done for *RS2* and *RS3* in similar manner. In the transformation, refactoring creates new synchronizations such as "*-lock*" and "*-release*" to preserve behavioral equivalence and redirects some synchronizations to the new processes. For example, "*-in_id1*" is redirected to *RS1*. In next sections, we will explain the transformations in more detail.

The refactored, new structure of the example system has some good properties which the original structure does not have. For instance, the highlighted region in Fig. 3 becomes tightly coupled inside but loosely coupled outside. The state-space size of (*S1*|*RS1*) is only 6 states/7 transitions (see last row of Table 1, where *d*={-*in_id1, out_ack, release*}).[4] Besides, the behaviors of the subsystem can be minimized more effectively because more rendezvous can be hidden inside the subsystem.

3. AN OVERVIEW OF REFACTORING

To refactor a process, the steps are to decompose its behaviors, make decomposed behaviors into new processes, and redirect communications to the new processes. In the meantime, behavioral equivalence must be preserved.

To explain how refactoring preserves behavioral equivalence, we introduce another process *P* into the example system. *P's* behavior is shown in Fig. 4, which invokes *in_id1, in_id2,* or *in_id3* iteratively and nondeterministically. When *R* is refactored, the shaded region *s1* in Fig. 2 is removed from *R* and wrapped into a new process *RS1*. The refactored behaviors and structure are shown in Fig. 5, where behaviors related to *S2* and *S3* are wrapped into *RS2* and *RS3* in similar manner but are not shown in the gure. After refactoring, *R* becomes a process containing two new synchronizations *-lock* and *-release*. In *P,* the action labeled *in_id1* is now replaced by two actions (*lock.in_id1*) and *in_id1* is redirected to *RS1*. In *RS1,* at the end of *ch1_ nish,* a *release* is added to signal *R* the end of an execution cycle in *RS1*.

In principle, the composite behaviors of *P,R,* and *RS1* must precisely simulate the behaviors of *P* and *R*. Let's consider several cases which could happen before refactoring: Suppose *P* invokes *in_id1* and returns. If *P* wants to invoke another *in_idi (i=1 to 3)*, *P* must wait until *R* nishes its sequence of synchronizations with *S1*. So, after refactoring, every *in_idi (i=1 to 3)* in *P* is guarded by a new synchronization *lock.* With *lock,* the new *P* is not able to invoke *in_id1 (i=1 to 3)* continuously. Only after *lock* is granted, new *P* can invoke *in_id1,* which is now redirected to *RS1.* In other words, *R's* new behavior is like a binary semaphore. It makes sure only one *in_idi (i=1 to 3)* in *RSi* is invoked at any time. Thus, the purpose of *release* is obvious. It is used by *RS1* to notify *R* that *RS1* has nished its execution cycle. *R* must be released to allow *P* to invoke another *in_idi (i=1 to 3).*

Mathematically, a behavioral equivalence is needed to justify the transformation. We resort to an equivalence that relates external behaviors of subsystems using weak bisimulation. For instance, we view original (*P*||*R*) as a subsystem because it is changed by the transformation. So, its interfaces are {*chi_start, -chi_wait, chi_ nish*}*, i=1 to 3.* Communications like *in_idi, i =1 to 3,* become internal actions of the subsystem and therefore should be restricted. After refactoring, (*P*|*R*) becomes (*P*|*R*|*RS1*|*RS2*|*RS3*). The external interfaces remain the same. The newly added synchro-

[4]The difference of size is not so signi cant in the example, because it is a very small system. For real applications, the difference can be enormous.

nizations, *lock* and *release,* are internal to the subsystem, therefore, should be restricted. So, in this example, the behavioral equivalence before and after the transformation can be formally expressed as

$$(P|R) \setminus \{in_idi, i = 1 \; to \; 3\} \approx$$

$$(P|R|RS1|RS2|RS3) \setminus \{in_idi, i = 1 \; to \; 3, release, lock\},$$

where "\" is the restriction operator and "≈" is the weak bisimulation of CCS. This equivalence can be checked by tools if the correctness of transformations is ever doubted.

As present, we borrow weak bisimulation to justify the correctness of our transformations because it is well-known and supported by several veri cation tools. Nonetheless, weak bisimulation is not capable of relating two systems for some properties, such as liveness. So, we are working on a new equivalence relation which can precisely relate two systems before and after refactoring. In principle, refactoring does not lose properties like liveness.

Some readers may interest in knowing why we favor CCS over CSP in refactoring. We use the example to explain. It is known that CSP rendezvous is of multi-way rendezvous semantics, which can be formally described as:

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P||Q \xrightarrow{a} P'||Q'}.$$

Suppose there is a process waiting to synchronize with a, it must wait for all other processes which can invoke a. The number of processes participating in the rendezvous may be greater than two.

On the other hand, CCS rendezvous is of two-way rendezvous semantics, which can be formally described as

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}.$$

So, in CCS, if two processes with a want to rendezvous with a communication $\bar{a}$ at the same time, the two processes compete for it. Because of this competition style of rendezvous, *R* can be as simple as that in Fig. 5. That is, the behavior of *R* is independent of other processes. However, in CSP semantics, if we want to have processes compete for some resources, we must introduce more communications to do so. If we adopt CSP semantics, *R's* behavior must be like Fig. 6(A). Its connection structure is shown in Fig. 6(B). The connections between *R* and other processes, unfortunately, grow as the number of *Si* grows. The structure is not as effective for utilizing compositional techniques compared with the one of CCS semantics. In addiction, the structure is inapplicable to inductive veri cation in [2].

4. REFACTORING TRANSFORMATIONS AND TOOL SUPPORT

To automate refactoring, we adopt Promela as our front-end language and add refactoring commands to its syntax. Promela is a popular design language due to the popularity of SPIN. We select a subset of Promela's syntax (e.g., excluding executable commands like *printf()*) and add some keywords for refactoring. The syntax is called *rc-Promela,* where "*r"* stands for *"refactoring"* and *"c"* stands for *"ccs."* We build a parser in *rc-Promela* syntax to generate CCS state graphs for Promela codes. At present, these CCS state graphs are used as input for Fc2tool[11]. Fc2tool is a toolsuite which can explicitly or implicitly explore state space under

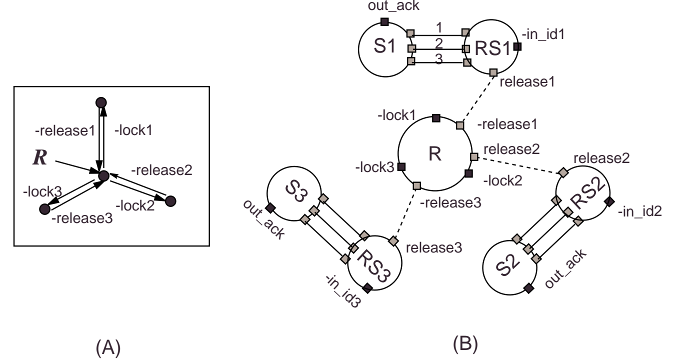


Figure 6: (A) The behavior of R if refactoring adopts CSP semantics. (B) The structure of example system if refactoring adopts CSP semantics.

CCS semantics. It also provides tools for minimizing and comparing state graphs by weak bisimulation, branch bisimulation, or strong bisimulation.

4.1 rc-Promela and segments

When *rc-Promela* parser is executed, it rst creates an abstract syntax tree (AST) for the Promela code. Next, we have an algorithm traverse the AST to generate CCS states and transitions repeatedly starting from an initial state. In the cases without refactoring, when we reach statement "*in?id*," the possible values of variable *id* are "symbolically expanded" to produce three transitions with labels *"in_id1", "in_id2",* and *"in_id3"* and three new states. The new states are put into a queue which saves the unexplored new states. Next, from each new state, one symbolically expanded value of *id* is used to traverse AST. The traversal continues until no more new states are generated and the queue is empty. The CCS state graph in Fig. 2 is so generated from its Promela code in the left.

To activate refactoring, users can use command "**refactorby { }**" to enclose a block of codes. For example, we can refactor *R* by enclosing its Promela codes as follows:

```
proctype R() {
 mtype id ;
 mtype waitmsg ;
 refactorby id {
  do
  :: in?id ->
     if
     ::id ==id1->ch1!start;
       ch1?waitmsg;ch1!finish ;
     ::id ==id2->ch2!start;
       ch2?waitmsg;ch2!finish ;
     ::id ==id3->ch3!start;
       ch3?waitmsg;ch3!finish ;
     fi
  od
 }
}
```

We call the enclosed block as *r-block*. The codes inside an r-block often begin with a statement which creates branches of control ow, such as *do* block followed by a "*in?id*" in this example or *if* statement. Decomposing behaviors at these locations often creates useful segments (de ned later) for compositional analysis.

When AST traversal algorithm enters an r-block, the way of generating CCS state graphs is changed. It begins to generate CCS

state graphs in *segments*. A *segment* is defined as the states and transitions generated by one pass of an r-block via the AST traversal algorithm. For the example above, starting from the first statement of r-block, which is "*in?id,*" we use *id1,* one of the symbolically expanded values of *id,* to traverse the r-block in one pass and we obtain a segment in Fig. 7.

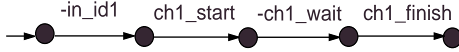


Figure 7: Segment seg_id1.

We call this segment *Seg_id1*. If we use other symbolically expanded values, *id2* and *id3* to traverse the r-block in one pass, two similar segments *Seg_id2* and *Seg_id3* are generated for this r-block. We call the first transition of a segment as *guard.*

4.2 Grouping segments

Once segments are generated, the next step is to divide the segments into groups and wrap each group in a new process. For process *R*, we already know *Seg_id1, Seg_id2,* and *Seg_id3* are divided into 3 groups and each is made into a new process (see *RS1* in Fig. 5). The grouping options are specified by the list behind the keyword **refactorby**. In this example, command **refactorby** is followed by a variable name *id,* which instructs the refactoring algorithm to group segments by every possible values of variable *id* (or conversely, divide the segments into groups by the values of *id*). For example, segments whose transition labels contain *id1* (which is symbolically expanded from *id*) are grouped together. Process *R* shows the simplest case of refactoring – one segment in one group. Section 3 already shows how it is transformed. In practice, process's behaviors can be more complicated. After grouping, one group may contain more than one segment. A unified transformation (see section 4.3) is derived to deal with such general cases.

We call the list behind keyword **refactorby** as *grouping options*. Since the segments in a group will be wrapped in a new process, the options decide the number of new processes to be created by refactoring. One mostly used grouping option is variable names like "**refactorby** *var1, var2*." This option first divides the segments into groups using all possible values of *var1* and next divides these groups again using all possible values of *var2.* Sometimes, for systems such as those in section 6 we need to use channel name and a variable name to divide the segments. The refactoring command is like "**refactorby** *ch, var,"* where *ch* is a channel name and *var* is a variable name. This option first divides the segments into two groups, one containing segments with *ch* in their edge label and none in the other group. Next, refactoring uses all possible values of *var* to divide the two groups into smaller groups.

To allow flexible refactoring decisions to be made, the grouping options can be specified as expression, such as "**refactorby** *ch1 and id == id1, waitmsg*." However, in practical applications we have encountered, most behavior patterns of processes are regular or patterned. So far, complicated options like that have never been used practically.

4.3 The unified decomposition transformation

In practice, behavior patterns to be refactored are often complicated by parameterization and the presence of data values. A variable with a finite range is typically "unrolled" in finite state verification, i.e., each state s in a process may be replaced by s_1, s_2, ... s_n for the n possible values of the variable (or the cross product of multiple variable values). On these states, same request may be responded differently. In Fig. 8, we introduce another process *RR*. *RR* receives an index from channel *in* and uses the index to address an element in array *cv*. Next, *RR* output the value of the element and flip the element's value.

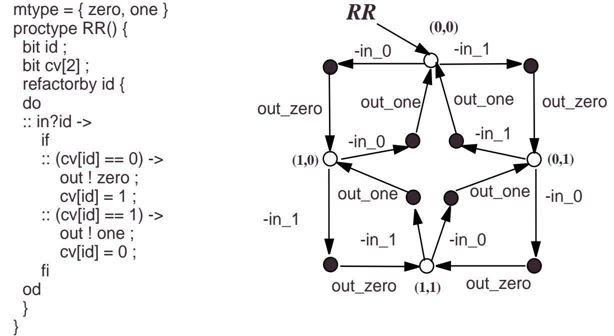


Figure 8: The Promela code and state graph of process *RR*.

When we begin traversing the Promela code to generate state graph, the initial state is set as a product of $(cv[0], cv[1])$, which is initialized as (0,0). The easiest way to represent a state is using the product of all variables plus an address of current statement. There are three variables $cv[0], cv[1]$, and id in this example but *id* can be excluded from the product because including it in the product is irrelevant for producing state graph. Whether a variable is relevant for producing state graph are checked statically by data flow analysis, but here we ignore the implementation details.[5] Also, in the figure, the address information in the product is omitted.

The state graph in Fig. 8 shows that when id=0 is received from channel *in* at first time, *RR* outputs "zero" and enters a new state (1,0). Next time, when id=0 is received, *RR* outputs "one" and returns to (0,0). So, when you send *RR* a zero, the outputs may vary depending on the state of $cv[0]$; that is, *RR* is a stateful process.

Suppose refactoring is activated. Using *id=0*, we begin the traversal of r-block. At the end of r-block, we produces a segment *Seg0* in Fig. 9. Let the traversal continue from a new state (1,0). We return to the beginning of r-block and use *id=0* again to traverse the r-block. Another segment *Seg1* in Fig. 10 is produced. Suppose the grouping option is "**refactorby** *id*." The two segments belong to the same group.

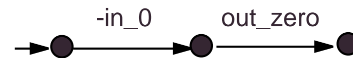


Figure 9: Segment $Seg0$.

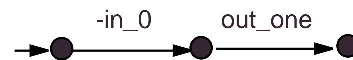


Figure 10: Segment $Seg1$.

[5] The data flow analysis is also used to decide which variable should be represented by a value process, which will be described later.

Consider wrapping *Seg0* and *Seg1* into a new process, say *RRS0,* and redirecting action *in_0* to it. One problem arises – should we redirect to *Seg0* or *Seg1*? We know it is determined by $cv[0]$. To assist *RRS0* making the choice, we introduce a new process called *Value Process* (VP). We use $VP(v)$ to denote the value process of a variable v. In Fig. 11 we show the value process of the variable $cv[0]$.

Constructing a $VP(v)$ for a variable v is straightforward. If there are n possible values of variable v, create n states to represent each value. At each state, add a transition which returns to itself labeled "*-v==#*," where # is the value of the state. Between states, if v can change its value from i to j, add a transition labeled "*-v:=j*" between state i and state j. Next, search in segments the places where v is updated and insert an edge labeled "*v:=#*," where # is the new value *v* is changed to. As of this example, we append "$cv[0] := 1$" to segment *Seg0* and "$cv[0] := 0$" to segment *Seg1*.

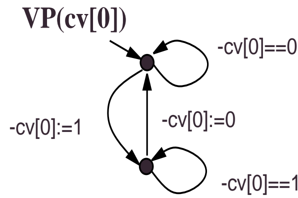


Figure 11: The value process of $cv[0]$

Once $VP(cv[0])$ is constructed, we place *Seg0* and *Seg1* together to create a new process *RRS0* (see Fig. 12). In the beginning, *RRS0* must be enabled by a new synchronization *"-startRRS0."* This synchronization is to prevent *RRS0* from rendezvousing with $VP(cv[0])$ privately. Next, either "$cv[0] == 0$" or "$cv[0] == 1$" is enabled by $VP(cv[0])$ to activate the correct segment. At the end of segment, $release$ is used to release *RR*. Note that inside the caller of (*in_0*), every (*in_0*) is now replaced by (*lock•startRRS0•in_0*).

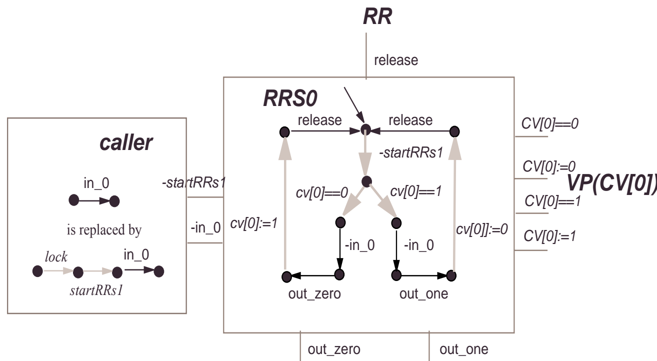


Figure 12: The state graph and interface of $RRS1$.

In Algorithm 1, we list the algorithm of this uni ed transformation. The algorithm has other variants to deal with different kinds of guard in segments, such as τ or else. These variants are not listed in this paper.

Without loss of generality, we assume there is only one state variable v which has n possible values. So, there are n self-loop transitions labeled "*-v==i*" in $VP(v)$. The algorithm assumes there are n segment to be wrapped into a new process. Let the segments be collected in a set α and each segment α_i is activated by transition "*v==i*." Let a be the guards of segments in α. The algorithm also assume transitions labeled "*v:=i"* have been inserted properly. The algorithm is quite straightforward. Its complexity is $O(N)$, where N is the number of segments.

Algorithm 1 The unified decomposition transformation

UnifiedDecomposition(α, a)
begin
  Construct $VP(v)$ for segment in α.
  Create an empty state graph T with an initial state s_0.
  Add transition $\langle s_0, -startT, s_1\rangle$ to T
  for each segment α_i in α do {
    copy α_i to T.
    add transition $\langle s_1, v == i,\ t_i\rangle$ to T, where t_i is the initial state of segment α_i.
    for every exited state s_e of α_i do
      add transition $\langle s_e, release, s_0\rangle$ to T.
  }
  update other processes whose edges labeled "a"
  into $lock.startT.a$.
end.

4.4 Simplifying state graphs

Although the processes in Fig. 12 look more complicated than the original, most synchronizations are internal between $RRS0$ and $VP(cv[0])$. This seemingly complicated synchronizations can be easily conquered by grouping them into subsystems. Fortunately, for many cases, we can transform them into a more compact form. For example, it is not dif cult to determine that segments $Seg0$ and $Seg1$ are activated one after another regularly in a loop. Using this observation, we can delete $VP(cv[0])$ and reduce *RRS0* into *RRS0'* of Fig. 13.

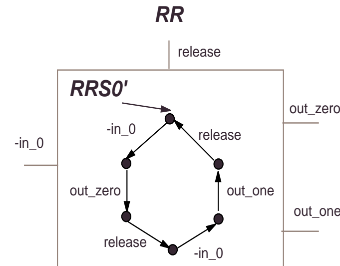


Figure 13: The simplified $RRS0$

Figure 14: The directed graph of Seg0 and Seg1.

To simplify a new process in this way we need to know whether segments activate other segments in a regular and predictable way. The algorithm is listed in Algorithm 2. The algorithm attempts to construct a directed graph representing the activation relations among segments. Let each node represent a segment. If only a directed edge is established from a segment a to a segment b, it means

segment b is activated for next time after segment a is executed. If there are more than one outgoing edges from a to other segments, it means one of those segments can be activated for next time. So, the directed graph of *Seg0* and *Seg1* can be constructed as Fig. 14. From *Seg0* to *Seg1* there is *exactly one* directed edge and vice versa. It means that *Seg0* and *Seg1* activate each other in a deterministic way and looped like the directed graph in Fig. 14. That is, they are eligible for simpli cation.

Let α be the set of segment from which the new process is constructed and $|\alpha| = n$. Let each segment α_i is activated by the transition labeled "*v==i*." Again, without loss of generality, we assume there is only one control variable v which have n possible values and each value i can activate segment α_i. Let $Pre(s)$ be the set of edges that start from some states and end at s. Let $Src(e)$ be a function which returns the source state of an edge e. Let $Exited(S)$ be the set of exited states of segment S.

Given a segment α_i, we use data ow analysis (procedure *ComputeGotoSet*) to compute the set of segments that could possibly be activated by α_i at every exited states. If a segment α_i can activate a segment α_j, we add a directed edge between node α_i and α_j. Initially, a segment activates itself. So, if v is not updated in the segment, there is at least one outgoing edge back to itself. Once the directed graph is constructed, we check if every node has exactly one outgoing edge. If not, the segments in α are not suitable for the simpli cation. If yes, we follow the directed graph G to connect the segments into a new process. The algorithm has a low-order polynomial complexity. For the procedure ComputeGotoSet, provided the number of edges into each state is bounded by a constant, the worst-case complexity is $O(S^2)$, where S is the number of states in a segment. So, the worst-case complexity of Algorithm 2 is $O(NS^2)$, where N is the number of segments.

Note that Algorithm 2 is always used to check segments rst. If they are eligible for simpli cation, follow its directed graph to connect segments. If not, Algorithm 1 is used to wrap them into a new process.

5. TOOL SUPPORT FOR COMPOSITIONAL ANALYSIS

To facilitate compositional analysis, we build a set of tools on top of Fc2tool [11] to automate hierarchical composition. Although Fc2tool provides some support for compositional analysis, it is too tedious and dif cult to use directly. For example, to create a hierarchy in Fc2tools, users must create, label, and connect every port by hand, which is error-prone and time consuming.

To use our tools to compose a system hierarchically, a user only needs to put the state-graph les (in a format for Fc2tools) in a directory and provide a hierarchy le like the following:

```
T1 := P R
T2 := S1 S2 T1
@ T2 is the whole system
```

Our tools will compute the necessary information automatically. In the example, suppose there are four state-graph les, *P,R,S1,* and *S2.* When P and R is composed into $T1$, our tools examine the directory and know its environment is constituted by $S1$ and $S2$. Correct label restriction (or exportation) for $T1$ is computed automatically. Unless speci ed, weak bisimulation is the default method for minimizing the state space of subsystem. In next section, all the experiments are done in this environment with 128M of memory under Linux platform.

Algorithm 2 The simpli cation transformation

```
Simplification(α)
begin
  Initialize G as an empty directed graph.
  Create a new node t_i for segment α_i in G.
  // construct directed graph G from segments
  For each segment α_i in α do
    goto ←ComputeGotoSet(α_i);
    for each integer k in goto do
     add a directed edge t_i → t_k in G
   if (|goto| == 1) then mark node t_i
  end for ;

  // check the directed graph to see if it is
  // eligible for reduction
  if (all the nodes in G are marked) then
    // the case is eligible for reduction
    Follow G to connect the segments
    into a new process.
  else
    return "not eligible for reduction";
  end if ;
end.

procedure ComputeGotoSet(α_i)
begin
  Let s_0 be the initial state of α_i;
  goto(s_0) = {i}; // initially, a segment activates
             // itself for next time.
  Set out(e) = { } for all the edges of α_i.
  Repeat
   for each state s in α_i do
     for each edge e ∈ Pre(s) do
      if (label(e)=="v:=i") then out(e) = {i};
      else out(e)=goto(Src(e));
     end for ;
     goto(s) := ∪_{e∈Pre(s)} out(e);
   end for ;
  Until goto(s) has no change for all s;
  return ∪_{s∈Exited(α_i)} goto(s);
end.
```

6. EXAMPLES

In this section we demonstrate the power of our approach by two examples. We choose the examples by two reasons. First, both examples have been used to gauge the scalability of veri cation tools. Second, when their system sizes increased, their as-built architectures make compositional techniques futile.

6.1 The elevator system

The model of elevator system is extracted from the elevator system of Richardson et al. [13]. Its implementation uses array of Ada tasks and is designed to be extended to arbitrary number of elevators. If the number of elevators is n, there are $n+3$ tasks, including n $elev_sim_task[i]$ which emulate the moving elevators for lifting customers, one *controller* task to command elevators to serve hall calls or car calls, one command dispatcher task (*elevator*), and one task (*driver*) to emulate customer pushing the hall call or car call button. In [5], Corbett analyzed (by global analysis) the system with maximum to 4 elevators. However, due to the difference in analysis tools used, memory capacity, and platform, we can only analyze up to 3 elevators in our environment.

We use elevator index and channel name as grouping options to refactor *controller* and *elevator*. In the model, they both are not stateful processes, so, no VP is created. The model is refactored in a way similar to the steps of refactoring the example in section

3. The number of new tasks created by refactoring plus the original task is listed in Table 2.

Fc2tools can enumerate reachable states explicitly and report the number of explored states and transitions. In a composition hierarchy, among all the subsystems analyzed, we pick the one which consumes most memory as the memory requirement to accomplish the compositional analysis. We compare three methods in Fig. 15. They are global analysis, compositional analysis without refactoring, and compositional analysis with refactoring. The hierarchy to compose the system without refactoring is (..((*controller*|*elevator)*| *elevator_sim_task1)*|..*elevator_sim_taskn)*| *driver)*. This hierarchy is carefully chosen by experience and trial-and-error so that state explosion is at least not worse than global analysis.

In the experiment, both global and compositional analysis without refactoring exhaust memory rapidly. On the other hand, the refactored model shows mildly linear growth of memory usage and the ability to analyze hundreds of elevators. The hierarchy for composing the refactored elevator system is to compose a base system with one elevator rst and then gradually add other elevators. Weak bisimulation and context constraints [3] are used to reduce subsystems in the hierarchy. One of the reasons that it can be analyzed to hundreds of elevators is that its refactored structure is "near to" a structure that is suitable for inductive veri cation (see [2]). We only check for deadlocks in this experiment. Because we adopt weak bisimulation, we currently limit our approach to safety properties, where a safety property can be translated into a deadlock detection problem [4].

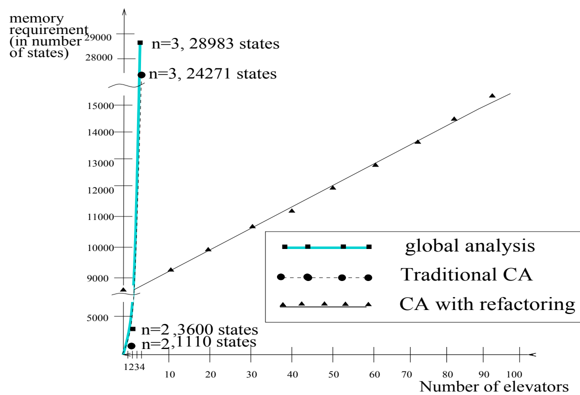


Figure 15: The states generated for elevator system by (1) global analysis (2) compositional analysis (3) compositional analysis with refactored structure.

6.2 Chiron user interface system

Chiron user interface system [10] is a moderate-size concurrent Ada program. It was built to address concerns of cost, maintainability, and sensitivity to changes in the development and maintenance of user interfaces for large applications. Chiron's design philosophy is to separate application code from user interface code. So, there are user interface agents called *artists* attached to selected data abstract types (ADT) belonging to the applications. At runtime, each artist can register *events* of interests to *dispatcher*. Whenever there is an operation call on the ADT, the dispatcher intercepts the call and noti es each of the artists associated with that ADT with the event.

Chiron has been analyzed by Young et al. [16] and Avrunin et al. [1], both with 2 artists analyzed. Its Ada code and Promela code can be accessed via *http://laser.cs.umass.edu/veri cation-examples*. In [1], different analysis tools (INCA,SPIN and FLAVERS) are stressed by increasing the event number of Chiron model. In that study, they decompose the dispatcher task into a subsystem with a separate task that maintains the array of each event, together with a single interface task that receives the requests for registration, unregistration, and the noti cation of events and passes them to the appropriate task for a particular event. Consequently, INCA shows better performance than SPIN and FLAVERS. The decomposition resembles our refactoring. However, it is done by rewriting design codes which requires human expertise and is dif cult to automate and guarantee behavioral equivalence.

To demonstrate the power of refactoring, our tool is stressed by increasing the number of artists. A 2-artist Chiron consists of 6 tasks. So, for an n-artist Chiron, there are $n + 4$ tasks.

When we began refactoring *dispatcher* task, we begin to realize why the number of artists has been limited to two in the past. For each event, the dispatcher maintains an array for bookkeeping the registered artists. The array is implemented as a queue. For example, suppose there are 3 artists, a_1, a_2 and a_3, registering an event e consecutively. Let the event array be $e[\,] = (a_1, a_2, a_3)$. If a_2 unregisters the event e, the content of $e[\,]$ becomes $(a_1, a_3, _)$; that is, the artists behind a_2 are shifted left by one element. Assume there are n artists, all possible combinations of the array are $1 + \sum_{i=1}^{n} \binom{n}{i} i!$. If there are m event array, the combinations are $(1 + \sum_{i=1}^{n} \binom{n}{i} i!)^m$. So, task dispatcher grows at a prodigious rate as the number of artists increased. On the other hand, for a 2-artist dispatcher, it contains only 5 combinations. When event number is m, the size of 2-artist dispatcher is proportional to 5^m and the number of tasks remains unchanged.

In our rst attempt, the uni ed transformation constructs a VP for each array element and wraps segments into new processes for different artists. The grouping options are channel name and artist id. Unfortunately, the refactored structure has little hope to scale well compositionally because VPs not only communicate with segments, but also with other VPs. This happens when an artist is unregistered. It starts a cascading changes of VPs because of shifting elements in the event array.

After a second look at the code, we discover that the event array, though implemented like a queue, is actually used only for keeping track of which artists are registered. The unregistration need not obey the FIFO rule. We are not sure why the event array is so implemented. Actually, a bit array of size n is adequate for the bookkeeping. For example, an event array $e[\,] = (1, 0, 1)$ means artists a_1 and a_3 have registered. If an artist wants to unregister the event, dispatcher simply sets its bit to zero. By replacing the queue with this bit array, the size of dispatcher becomes proportional to 2^n. Although 2^n is still a formidable growth rate, refactoring can create VPs and new tasks which are loosely coupled to its environment. We analyze the refactored Chiron with bit array compositionally. It can be analyzed up to 14 artists as in Fig. 16.

Table 2: The summary of refactored elevator model.

task name	no. of states	the number of subtasks after refacting
controller	$21n$	$2n + 1$
elevator_sim_task[i]	8	no change
elevator	$7n$	$n + 1$
driver	$8n$	$n + 1$

Figure 17: The refactored structure of 2-artist Chiron system.

Figure 16: The states generated from refactored Chiron (bit array version).

The refactored structure of a two-artist Chiron is shown in Fig. 17. The dispatcher task is in the middle. The new tasks generated from decomposing dispatcher is named "*dispatchersubaXeY*," where X is the index of artist and Y is the index of event. The structure may look complicated in the beginning. Searching a tractable composition hierarchy seems difficult. However, heuristics exist for searching the hierarchy.[6] We can search in the components of first artist first. Like composing the elevator example, we start by composing all the components of artist *a1*. Let A = (notifye1| notifye1suba1), B = (notifye2 | notifye2suba1), C = (dispatchersuba1e1|VPa1e1), D = (dispatchersuba1e2|VPa1e2) , E = (adtwrapper| clientinit | artistmgr). The composition hierarchy we use is ((((C | a1 | dispatcher) | D) | A| B) | E). After the base system is completed, we begin composing components of artist *a2* into the base system.

Composing the base system always generates maximum states and transitions, because there are transitions and states must be exported for rendezvousing with other artists later. The number of these exportations is strictly decreased when more artists are composed. In the experiment of 15 artists, the weak bisimulation of Fc2tool consumes too much time and disk space while composing the base system.

Although we consider the result a significant improvement, the extent of improvement is not as good as elevator system. The cause is that Chiron contains a FOR loop that makes rendezvous with all artists sequentially. This small part of behaviors can not be refactored perfectly and made into a semaphore independent of other processes. As a result, more exportations are inevitable. Since FOR loop like that can be common in practice, research to address the problem continues. We have found that if the order of making rendezvous in a FOR loop is irrelevant to property of interest, we may refactor the behavior into a better structure for compositional analysis. However, more works remain to be done for that, so we exclude the results from this paper.

[6]Finding an optimal solution is NP-hard, but in practice an acceptable solution can often be found with modest effort.

7. RELATED WORK

Researches addressing systems that are not suitable for compositional, incremental analysis can be found in [15, 6]. Corbett and Avrunin [6] observe that in the analysis of large and complex programs, a module I may not be further decomposed into many loosely coupled units and the composition of its processes may yield intractable results. They assume the module interfacing with its environment is a specification process S. The analysis of the module becomes a problem of proving the trace equivalence between S and I using integer programming techniques. Their approach, however, is limited to a restricted class of systems, requiring processes to be deterministic.

Yeh and Young [15] considered the problem of "design for analysis" with a goal of compositional analysis using process algebra. In that work, it was proposed that analyzability should be an important factor in structuring the design of actual implementations. Be that as it may, one is often in the position of trying to apply analysis post hoc to systems that are not structured as we would like. Even if one has the luxury of structuring a design with analysis in mind, there may be good reasons (performance, physical distribution, reuse of existing artifacts) for the "as built" system to differ from the structure needed for verification, and a way of reconciling the verified and as-built structures will still be needed.

8. DISCUSSIONS AND CONCLUSIONS

In [9], Holzmann has argued that blindly derived design models are unlikely to be amenable to analysis. That argument applies to other verification tools and may apply to refactoring on occasion as well. The array usage as a queue in Chiron is an example. That particular data structure may prevent refactoring from constructing suitable structures for compositional analysis. Fortunately, behaviors of most systems are regular or patterned. The growing popu-

larity in design patterns shall reduce the occurrence of those problematic behaviors. On the other hand, we will continue exploring new transformations for those problematic behaviors and data structures.

Automating the heuristic search of a tractable hierarchy is another problem worth pursuing. Furhter automation is important for producing refactoring tools that can be used routinely by software developers.

In summary, the refactoring transformations described here permit decomposing processes and recombining them in a structure that is more amenable to compositional analysis using process algebra, allowing larger versions of a model to be veri ed. We have described some important refactoring transformations and a refactoring tool that automates restructuring models in a subset of Promela.

9. REFERENCES

[1] G. S. Avrunin, J. C. Corbett, M. B. Dwyer, C. S. Pasareanu, and S. F. Siegel. Comparing nite-state veri cation techniques for concurrent software. Technical Report UM-CS-1999-069, Dept. of CS, University of Massachusetts, November 1999. (in preparation).

[2] Y. Cheng. Refactoring design models for inductive veri cation. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA2002)*, pages 164–168, Rome, Italy, July 2002.

[3] S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4):334–377, October 1996.

[4] S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 8:49–78, January 1999.

[5] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 2(3):161–180, March 1996.

[6] J. C. Corbett and G. S. Avrunin. Towards scalable compositional analysis. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering.*, pages 53–61, New Orleans, Louisiana, USA, December 1994.

[7] S. Graf and B. Steffen. Compositional minimization of nite state systems. In *Proceedings of the 2nd International Conference of Computer-Aided Veri cation*, pages 186–204, 1990.

[8] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, NJ 07632, 1991.

[9] G. J. Holzmann. Designing executable abstractions. In *Proceedings of the second workshop on formal methods in software practice*, pages 103–108, Clearwater Beach, Florida USA, March 1998.

[10] R. K. Keller, M. Cameron, R. N. Taylor, and D. B. Troup. User interface development and software environments: The Chiron-1 system. In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 208–218, Austin, TX, May 1991.

[11] E. Madelaine and R. de Simone. *The FC2 Reference Manual*. Available by ftp from cma.cma.fr:pub/verif as le fc2refman.ps, INRIA.

[12] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.

[13] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Speci cation-based test oracles for reactive systems. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 105–118, Melbourne, Australia, May 1992.

[14] W. J. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proceedings of the Symposium on Software Testing, Analysis, and Veri cation (TAV4)*, pages 49–59, Victoria, British Columbia, October 1991. ACM SIGSOFT, ACM Press.

[15] W. J. Yeh and M. Young. Re-designing tasking structure of Ada programs for analysis:A case study. *Software Testing, Veri cation, and Reliability*, 4:223–253, 1994.

[16] M. Young, R. Taylor, D. Levine, K. A. Nies, and D. Brodbeck. A concurrency analysis tool suite for Ada programs: Rationale, design, and preliminary experience. *ACM Transactions on Software Engineering and Methodology*, 4(1):65–106, Jan 1995.

Flow Equations as a Generic Programming Tool for Manipulation of Attributed Graphs

John Fiskio-Lasseter and Michal Young[*]
Department of Computer &
Information Science
University of Oregon
Eugene, OR 97403
{john ,michal }@cs.uoregon.edu

ABSTRACT

The past three decades have seen the creation of several tools that extract, visualize, and manipulate graph-structured representations of program information. To facilitate interconnection and exchange of information between these tools, and to support the prototyping and development of new tools, it is desirable to have some generic support for the speci cation of graph transformations and exchanges between them.

GENSET is a generic programmable tool for transformation of graph-structured data. The implementation of the GENSET system and the programming paradigm of its language are both based on the view of a directed graph as a binary relation. Rather than use traditional relational algebra to specify transformations, however, we opt instead for the more expressive class of o w equations. Flow equations—or, more generally, systems of simultaneous xpoint equations—have seen fruitful applications in several areas, including data and control o w analysis, formal veri cation, and logic programming. In GENSET, they provide the fundamental construct for the programmer to use in de ning new transformations.

Categories and Subject Descriptors

D.1.0 [**Programming Techniques**]: General; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.3.2 [**Programming Languages**]: Language Classi cations— *Specialized application languages,Constraint and logic languages*

General Terms

Languages, Veri cation

[*]Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0620 and F30602-97-2-0034. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for pro t or commercial advantage and that copies bear this notice and the full citation on the rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior speci c permission and/or a fee.
PASTE'02, November 18–19, 2002, Charleston, SC, USA.
Copyright 2002 ACM 1-58113-479-7/02/0011 ...$5.00.

1. INTRODUCTION

Many problems of software analysis can be usefully modelled by viewing the structure of the problem data as a directed, attributed graph and de ning analyses and transformations on this structure. Consequently, the past three decades have seen the creation of several tools that extract, visualize, and manipulate graph-structured representations of program and design information, for applications spanning a broad range of elds. This includes, on the one hand, specialized programs such as data/control- o w analyzers for optimizing compilers and model checkers. On the other hand, it includes a number of tools designed for more general tasks of program analysis, such as reverse engineering, program comprehension, design, and visualization.

In reverse engineering, for instance, *fact extractors* generate raw information about a software system (call graphs, directory structure, etc.). These are ultimately displayed by *visualization tools* such as AT&T's `dotty`, but only after considerable processing to elide detail, as well as transformation to the graph notation supported by the tool.

In a different, hypothetical domain (although the example is inspired by a real veri cation method of de Alfaro [5]), we might use a *web crawler* to traverse the pages of a web site, storing the results as a crawl-graph. If the crawler categorizes web pages according to, say, access control attributes, we can inspect the displayed crawl graph to look for security violations of private-data pages (in the form of paths from the home page to private pages that do not go through control pages). Ideally, we could use a tool that would transform the crawl graph so that insecure browsing paths were clearly displayed by a graph viewer.

Three themes run through these examples. First is the use of multiple graph-based tools in combination, for analyses ranging from established industrial practices to experimental techniques. With this comes the second theme: Each composition of tools is predicated on the ability of one tool's input format to be compatible with the other's output. The third theme is displayed in the web crawler example by the wish for an automatic display of security violations: the occasional need for rapid creation of a hitherto unconceived analysis. Taken together, these themes underscore the following: To support the prototyping and development of new tools, and facilitate the interconnection and exchange of information between existing tools, *it is desirable to have some generic support for the speci cation of graph transformations and exchanges between them*.

We believe that o w equations provide an attractive generic technique for programming such transformations. Viewing the typed

edges and attributes of a digraph as binary relations, we can de- ne a system of simultaneous xpoint equations whose solution is a set of new relations corresponding to the edges and attributes of the desired transformed graph. This approach gives a nice combination of declarative programming character, expressive power, sound theoretical foundations, and modest implementation cost.

To explore this approach, we have created GENSET, a generic programmable tool for the transformation and exchange of graph-structured data using o w analysis. At the heart of GENSET is an interpreter for a domain-speci c language of o w equations in which the desired graph transformations can be programmed. Both the implementation of the GENSET system and the programming paradigm of the GENSET language are based on the view of a directed graph as a binary relation, a viewpoint that is extended to the attributes on the nodes of the graph.

The remainder of this paper is organized as follows. In the next section we give an overview of the GENSET language. This is followed in §3 by a discussion of some interesting aspects of our implementation. We follow this in §4 with a discussion of the possible applications for a tool such as GENSET. A demonstration of our approach on a real example is given in §5.Finally, we survey related work in §6. Possibilities for future work are offered in the conclusion of §7.

2. GENSET—AN OVERVIEW

GENSET is best thought of as a "little language": a compact, special-purpose, declarative language designed speci cally as a utility for programming transformations of edge-typed, directed, attributed graphs. Such graphs are the primary data value in GENSET, and the only kind of value a GENSET script produces.

Graphs in GENSET are de ned over a single, nite universe of untyped nodes, called *items*. Items are de ned inductively to be either *atoms* (roughly as in Lisp) or *pairs* of items. Edges may be thought of as ordered pairs (src, tgt) of items, directed from src to tgt. Every edge in a graph has a *type* T, out of a nite set of edge types. Equivalently, we can view each edge type T as a graph with label T.

As with graph transformation approaches based on relational algebra [9, 11], the graphs input to and constructed by a GENSET program are represented internally as a collection of binary relations, one for each edge or node attribute type. The edges of a graph T are then represented as pairs belonging to relation T. A node may also have a nite number of *attributes*, and every attribute may itself be understood as a relation. For example, a node n with $Color$ attribute of red is represented by the edge (n, red) in the $Color$ relation.

The basic programming construct in GENSET is a block of *ow equations*: simultaneous equations over a rst-order predicate calculus, extended with xpoint operators. Although syntactically similar to iteration in an imperative language, there are important conceptual differences.

The general form can be illustrated by this example:

```
for x in IterExp_x, y in IterExp_y do
  R(x) := least E_R;
  S(y) := most E_S;
od;
```

Each "statement" can be understood as an equation de ning a new relation: the expression on the right-hand side is used to compute, for each item a in the relation's domain, the set of items in the relation's image to which a maps. The domain from which a is drawn is the value of the corresponding $IterExp$ expression in the initialization of the block. In the above example, the equation `R(x)` := `least` E_R; de nes the relation

$$R = \{ (a, b) \mid a \in v_x, b \in v_R \}$$

where v_x and v_R are the values that result from evaluating the expressions $IterExp_x$ and E_R, respectively.

The scope of an "iterator variable" such as x is limited to the right-hand sides of the equations in which it appears on the LHS. For example, x is in scope in the expression E_R, while y is not. On the other hand, the scope of identi ers denoting relations is the entire GENSET script.

Note that `R(`x`)` can occur recursively in the equation E_R that de nes it, and that R and S can be mutually recursive. In contrast with ordinary iteration, this means that each "statement" in a for-block is evaluated *at least* once for each element in the value of its controlling iterator expression, but it may be re-evaluated as many times as necessary to reach a *xed point*. A ner point here is that `for`-blocks are themselves evaluated in source code order, and so forward references—references to relations that are de ned by blocks following the current one—are disallowed. This avoids the problem of mutually recursive de nitions inter-block, allowing each block of equations to be individually iterated to a solution.

Whether the least or greatest solution is calculated for an equation depends on which of the keywords, `least` or `most`, is given at the beginning of the RHS. All equations in GENSET are quali- ed by one of these two *xpoint operators*, which extends over the entire RHS of an equation. If no operator is speci ed, the default choice is `least`. These keywords function like the μ and ν operators of the μ-calculus [15], but, for simplicity, they are limited to one use per equation: each one quanti es its whole RHS and they cannot be nested. For the least xpoint of an equation, the value of the corresponding LHS is initialized to the empty set. For the greatest xpoint, the LHS is initialized to the "universe," a value that must be implemented with some care. See §3.3 for a discussion of this issue and some of the restrictions that it carries.

Genset expressions evaluate to sets of items. The syntax is given by seven general classes in Figure 1.

Direct set construction is limited to the empty and singleton sets, as well as some limited uses of the identi ers denoting relations (as $filter$ or u_op operands). When the `single` keyword is applied to a variable x, the value is the singleton set consisting of the node to which x is currently bound. If applied to the constant `"x"`, the value is the set $\{x\}$. Node constants have global scope, and are created on-the- y if necessary.

The *binary operations* are set union, intersection, difference, and cross-product, each with the expected meaning. *Unary operations* are de ned only on an expression e whose value v_e is a set of pairs (*i.e.*, a relation); they facilitate the extraction of a relation's domain, image, or both. One can also select a subset of the pairs in a relation r, using a lter expression e on either the domain or image of r.

The two *appl* constructs are borrowed from relational algebra approaches (see [11], for example) for their utility: in the traversal of a graph r, we will often need to know the successors (resp. predecessors) of a node x. From the relational viewpoint, this corresponds to a projection of x through r (or vice-versa), an operation that we will term relation *application* (resp. *inverse application*). Relations can be applied either to variables or (by enclosing the identi er in `" "`) node constants.

The *combination expressions* are (very) loosely modeled on the syntax of the reduction operator `/` of APL, but are more similar in intent to the application in data o w analysis of a "combination operator" for combining the o w information collected along dif-

e	:=	$direct \mid b_op \mid u_op \mid filter$ $\mid appl \mid combine \mid cond$	*(expressions)*
$direct$	:=	`null`	$\emptyset$
		| `single("x")`	$\{x\}$, x a constant
		| `single(`x`)`	$\{n_x\}$, where n_x is the current binding for x
		| r	$\{(x, x') \in r\}$, r a relation
b_op	:=	e `union` e	*(set union)*
		| e `intersect` e	*(set intersection)*
		| e `-` e	*(set difference)*
		| e `cross` e	*(cross product)*
u_op	:=	`dom` e	$\{ x \mid \exists x'.(x, x') \in v_e \}$
		| `img` e	$\{ x' \mid \exists x.(x, x') \in v_e \}$
		| `base` e	$($`dom` $e) \cup ($`img` $e)$
$filter$	:=	r `ofdomain` e	$\{ (x, x') \mid (x, x') \in r \wedge x \in v_e \}$
		| r `ofimage` e	$\{ (x, x') \mid (x, x') \in r \wedge x' \in v_e \}$
$appl$	:=	$r(x) \mid r($`"x"`$)$	$\{ x' \mid (x, x') \in r \}$
		| $_r(x) \mid _r($`"x"`$)$	$\{ x' \mid (x', x) \in r \}$
$combine$	:=	`/union` x `in` e_1 `:` e_2	$\bigcup_{x \in e_1} e_2$
		| `/intersect` x `in` e_1 `:` e_2	$\bigcap_{x \in e_1} e_2$
$cond$	:=	`if` p `then` e_1 `else` e_2 `fi`	*(conditional evaluation)*

Figure 1: Summary of GENSET constructs

ferent paths. With either operator, the scope of variable x is the expression e_2.

Finally, GENSET includes a construct for *conditional evaluation* of an expression, based on the value of predicate p. Support is available for predicates that test emptiness of a set and set containment/comparison, and predicates may be combined using the standard propositional boolean connectives (`not`, `and`, `or`).

To illustrate, suppose we have extracted from a program the control o w graph `Flow` along with suitable `Kill` and `Gen` relations. We can use these to construct, for example, the classical *reaching de nitions* analysis[2]:

```
for x in (base Flow) do
  RD(x) := /union w in _Flow(x):
             (RD(w) - Kill(w)) union Gen(w);
od;
```

which is notation in GENSET for the familiar textbook o w equation

$$RD(x) = \bigcup_{w \in Flow^{-1}(x)} (RD(w) \setminus Kill(w)) \cup Gen(w)$$

Another example, using the `most` operator, is the computation of dominators. For any node n in the $Flow$ graph, the *dominators of n* are those nodes which must be traversed on any path from the root of $Flow$ to n:

```
for x in (base Flow) do
  DomsOf(x) :=
   most (if (empty _Flow(x)))
          then single(x)
          else (single(x) union
            (/intersect y in _Flow(x): DomsOf(y)))
         fi);
od;
```

This example also demonstrates the use of an `if-then-else` expression. The standard dominators equation is de ned by a pair of simultaneous equations, the choice of which depends on whether x is the root node of $Flow$.

3. IMPLEMENTATION

Implementation of an interpreter for the GENSET language involved several interesting challenges. We summarize the main features in this section.

3.1 Generic Worklist Algorithm

Interpretation of a GENSET program takes the form of an iterative data o w analysis [13], using a worklist algorithm:

$$
\begin{array}{l}
while\ (Worklist\ not\ empty) \\
\quad (EQN, a) \leftarrow Worklist.extract() \quad // \texttt{R(a) :=} e; \\
\quad e \leftarrow EQN.getRHS() \\
\quad R \leftarrow EQN.getLHS() \\
\quad v \leftarrow eval(e, a) \\
\quad if\ (R(a) \neq v)\ then \\
\qquad \forall\ dependent\ (equation, item)\ pairs\ (Q, b) \\
\qquad\quad Worklist.add(Q, b) \\
\qquad R(a) \leftarrow v
\end{array}
$$

Note that we perform a *nonin ationary* update, $R(a) \leftarrow v$ (instead of $R(a) \leftarrow v \bigsqcup R(a)$) [1]. Furthermore, this update is performed on *any* change to $R(a)$. As a consequence, the existence of a x ed point for any equation block, and hence termination of the worklist algorithm, can only be guaranteed by the equation itself.

The usual approach to ensuring termination is to guarantee that each equation is *monotonic*, from which the existence of a x-point follows. Although there are ways to give a static guarantee of monotonicity, or to enforce it at runtime, the current implementation of GENSET does neither. Instead we lay the responsibility for

termination of evaluation in the programmer's hands. The reason for this lies in the desire to provide as much e xibility to the programmer as possible. Divergence is not a desirable trait, of course, but a requirement of monotonicity placed on every equation would eliminate some analyses that nonetheless have xpoint solutions. One example of this is the *partial dead code elimination* of Knoop *et al* [14]. Their algorithm involves an analysis based on functions that individually are not themselves monotonic, but when considered as a family enjoy weaker properties suf cient to guarantee the existence of a xpoint solution [10].

3.1.1 Variable Dependencies

The other departure from the usual worklist algorithm lies in the determination of dependent $(equation, item)$ pairs to requeue—*i.e.*, those equations whose value might change because of the new value v. The intuition here is that when re-evaluation of an equation R results in a new value, the change will affect every equation Q whose right-hand side depends on R. More to the point, we do not need to re-evaluate anything else, and so can avoid a complete re-evaluation of the equation system. Although this optimization does not offer an improvement in worst-case complexity, in practice, it can produce signi cant performance increases.

When the set of equations is x ed syntactically (*e.g.*, the μ-calculus), dependence consists of one or more occurrences of R in the RHS of Q, and can be determined entirely from the source code. But in the case of data- o w equations (and also logic programming), we have the added challenge that the equations are parameterized over one or more variables, de ning a *schema* of equations. For example,

$$\texttt{for } x \texttt{ in } E_1 \texttt{ do ... R(x) :=} E_2\texttt{; ... od;}$$

de nes one R equation for each element of the value of E_1. Thus, when some $R(x)$ changes value, we need to know not just that some of the equations de ned by $R(x)$ could be affected, but which ones. If this cannot be determined, we must adopt the conservative approach of re-evaluating R on *every* node in the value of E_1.

While this necessitates a dynamic component to dependency determination, in traditional bit-vector data o w analyses, the *form* of the analysis is always the same. When, for some node a, the value of $R(a)$ changes, the affected equations are those corresponding to the adjacent nodes (either successors or predecessors) in the program's control o w graph, such as `Flow`, in the RD example above: $\{b \mid b \in \texttt{Flow}(a)\}$. In imperative programs, this graph is x ed before analysis and remains static throughout.

A GENSET script, however, represents an even more general case. While the language can be used to formulate bit-vector analyses over x ed graphs, this is not a necessary assumption. An equation block's iterator expression, for example, can be any legal GENSET expression, not just the nodes in the (pre-computed) `Flow` graph, and there is no requirement that the RHS of an equation combine information from successors or predecessors in `Flow` or any other single relation.

Our approach is based on a static analysis of the GENSET source code. During construction of the abstract syntax tree, we associate with each equation de nition R, a list of $(equation, expression)$ pairs, (Q, d_R^Q), one for each equation de nition Q in the same block as R. As mentioned in §2 above, each block is independently iterated to a x ed point, so no equations de ned outside the block can be affected by a change to R. Moreover, in practice the number of equation de nitions in a block is likely small, so this overhead, although quadratic in the number of equations de ned in the block, should be manageable.

The expression d_R^Q (not to be confused with a GENSET expression) represents the computation necessary to determine the items on which Q will need re-evaluation, when the value of $R(a)$ changes for some item a: this set is given by $d_R^Q(a)$. Its construction is essentially a partial evaluation of the RHS of Q. For an equation de nition $R(x)$:= E_R, the dependency expression for a de nition $Q(y)$:= E_Q (de ned over iterator domain Y), is the function $d_R^Q = \lambda x.dep(R, E_Q, x)$. $dep(R, E_Q, x)$ is de ned inductively on the structure of E_Q as in Fig. 2. Note that for the classical form

```
A(x) := /<op> w in _Flow(x):
              (A(w) - Kill(w)) union Gen(w)
```

we have

$$dep(A, \texttt{(A(w) - Kill(w)) union Gen(w)}, x) = \{x\}$$

and so the dependency expression is the special case

$$\lambda x.(Flow(x) \cup dep(A, \texttt{_Flow(x)}, x)) = \lambda x.(Flow(x) \cup \emptyset)$$
$$= \lambda x.Flow(x)$$

3.2 Set and Relation Data Structures

As discussed above, the edges of each graph (or equivalently, the members of each edge type in a graph) are represented in GenSet as elements of a binary relation. Since relation application and inverse application are perhaps the most commonly used expressions in a GENSET script, it is important that these operations be as fast as possible. Internally, relations consist of two hash tables, one for the forward and one for the inverse image. Each entry in the forward (*resp.* inverse) table corresponds to an item in the domain (*resp.* image), and it is stored in the table with a set of the items in the image (domain) to which it is related.

Ordinary sets are implemented using the Java `HashSet` class, which maximimizes the speed of insertion and retrieval of elements. One cost for this choice is that we are committed to the generic semantic view of nodes as being unordered. Except for equality, we cannot compare one node with another, nor can we easily choose any particular iteration order over elements (*e.g.* reverse postorder traversal).

3.3 "Infinite" Sets

The speci cation of a greatest xpoint equation (using the `most` keyword) requires that we have some way to initialize the equation to the "universe" value and perform set operations (*e.g.*, set difference) on this value. Unfortunately, it will not do simply to use the set of nodes contained in input graphs, nor even adding to this the set of named constants in the source code. On a pragmatic level, even if this top value could be determined in advance, it may be very large and hence impractical to use directly, if we can avoid it.

A more signi cant problem is that in many cases, it is the wrong value to use. The problem arises from the cross-product operator, as in this (silly but illustrative) example:

```
for x in (single("a") union single("b")) do
  S(x) := most (S(x) intersect
                (single("c") cross single("d")));
od;
```

This should give the relation $\{(a, (c, d)), (b, (c, d))\}$, but if we iterate from the universe $\{a, b, c, d\}$, we will instead get $S = \emptyset$. The point here is that while the number of *atoms* is x ed by the graphs given as input, the number of *items* is not. As a consequence, the universe in general cannot be statically determined: it is nite but of indeterminate size.

$$
\begin{array}{lll}
dep(R,\texttt{null},x) & = & \emptyset \\
dep(R,\texttt{single(}_\texttt{)},x) & = & \emptyset \\
dep(R,Q',x) & = & \begin{cases} \text{Y} & \text{, if } Q = Q' \\ \emptyset & \text{, if } Q \neq Q' \end{cases} \\
dep(R,u_op\ e,x) & = & dep(R,e,x) \\
dep(R,e_1\ b_op\ e_2,x) & = & dep(R,e_1,x) \cup dep(R,e_2,x) \\
dep(R,e_1\ filter\ e_2,x) & = & dep(R,e_1,x) \cup dep(R,e_2,x) \\
dep(R,r(y'),x) & = & \begin{cases} \{\text{x}\} & \text{, if } r = R \\ \emptyset & \text{, otherwise} \end{cases} \\
dep(R,_r(y'),x) & = & \begin{cases} R(x) \cup \langle\ old\ value\ of\ R(x)\ \rangle & \text{, if } r = R \\ \emptyset & \text{, otherwise} \end{cases} \\
dep(R,\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{fi},x) & = & dep(R,p,x) \cup dep(R,e_1,x) \cup dep(R,e_2,x) \\
dep(R,/op\ w\ \texttt{in}\ e_1 : e_2,x) & = & \begin{cases} dep(R,s(y),x) & \text{, if } e_1 \equiv s(y) \text{ and } dep(R,e_2,x) = \emptyset \\ dep(R,_s(y),x) & \text{, if } e_1 \equiv _s(y) \text{ and } dep(R,e_2,x) = \emptyset \\ _s(x) \cup dep(R,s(y),x) & \text{, if } e_1 \equiv s(y) \text{ and } dep(R,e_2,x) \neq \emptyset \\ s(x) \cup dep(R,_s(y),x) & \text{, if } e_1 \equiv _s(y) \text{ and } dep(R,e_2,x) \neq \emptyset \\ \text{Y} & \text{, otherwise} \end{cases}
\end{array}
$$

Figure 2: Static determination, for equation $Q(y) := E_Q$, of the nodes on which E_Q must be re-evaluated

Our approach is to treat the universe as if it were in nite. For such a value, we use a "symbolic in nity" by working instead with the *co-enumeration* (*i.e.* the complement) of an in nite set.

However, this itself raises another problem. Nontermination of expression evaluation is a separate consideration from the divergence that results when no solution exists for an equation. Unlike the absence of a x ed point, divergence of the expression evaluator cannot be considered an acceptable possiblity—it would be a bug in the interpreter itself, not the programmer's code. In the case of a co nite set, this means that we must be careful about enumeration of any value. In particular, the termination of expression evaluation with co-enumerated in nite sets requires some restrictions on the possible forms of a relation: for every set, we must be able either to enumerate the set itself, or else its complement. Moreover, when evaluation of a `for`-block has been iterated to a xpoint, each relation de ned in the block must be nite. Further restrictions prevent the possiblity of enumerating an in nite set (the initializations of all iterators must evaluate to nite sets), or of taking the cross product with an in nite set operand. As with all other typing properties in GENSET, these restrictions are checked dynamically.

4. APPLICATIONS

Although GENSET has enough expressive power to write equations for data o w analysis or CTL properties, it was designed for neither optimizing compilation nor model checking, and is not intended to compete with more specialized tools used in these areas (*e.g.*, SUIF [25]). Indeed, as it lacks the optimizations that render tractable the storage and rapid traversal of enormous control o w or state space graphs, it is likely unsuitable for either domain.

GENSET is intended rather to ll a niche similar to the role that Awk plays in the world of Unix text streams: The use of o w equations as a speci cation medium enables the programmer to draw from a class of graph manipulations which are too complex for simpler, less e xible tools, and with signi cantly less programming effort than would be required to write a special-purpose tool in, say, C or Java.

Hopefully, the near future will see the adoption of a standard graph exchange format for analysis and visualization tools; for example, the graph-based GXL format for XML [12]. In the presence of such a standard, we foresee a natural application for GENSET in the construction of "pipe- tting" components that facilitate the composition of off-the-shelf tools. In support of this goal, we designed GENSET to be as e xible as possible with respect to support of new exchange formats. The current implementation includes support for RSF (Rigi Standard Format) and AT&T "dot" format, as well as the early untyped form of GXL described in [12]. We are currently working to extend support to include the recent GXL changes.

5. EXAMPLE

Flow equations have traditionally been associated with problem domains in compilers and logic programming. To illustrate their utility as a basis for graph transformation tasks relevant to the analysis of programs and software systems, we apply our approach to a transformation known in the reverse engineering community as *lifting*.

As a basis for our analysis, we chose the Linux kernel example from the PBS Guinea Pig repository [18]. From this, we chose as an input graph a selection of edge types from the raw example: 3 relations used in the calculation (`contain`, `funcdef`, and `sourcecall`), and one that did not play a part. GENSET does not yet have a parameterization mechanism for input arguments[1], so these are read into a global symbol table, along the lines of `extern` variables in C. Edges we actually used represented 8136 function de nitions, 2068 source calls, and 1149 containment edges (directory structure). There were edges included in the le but not used in the transformation in the form of 6749 `include` edges.

A lifting transformation is used to produce a high-level view of a relation. This is done by lifting a relation between low-level entities (such as the original function to function call graph stored in the factbase by the `sourcecalls` relation) to a more abstract version of the relation, between higher-level entities (such as the directory to directory level). Those entities considered at the top level are represented as edges in the `toplevel` relation. The hierarchy is given here by the `contain` relation, which in the original factbase described containment of les in directories as well as directory nesting.

In the usual form of lifting, the `toplevel` relation would consist of exactly those entities that are at or above some level in the

[1]This is under development. See the comments in §7.

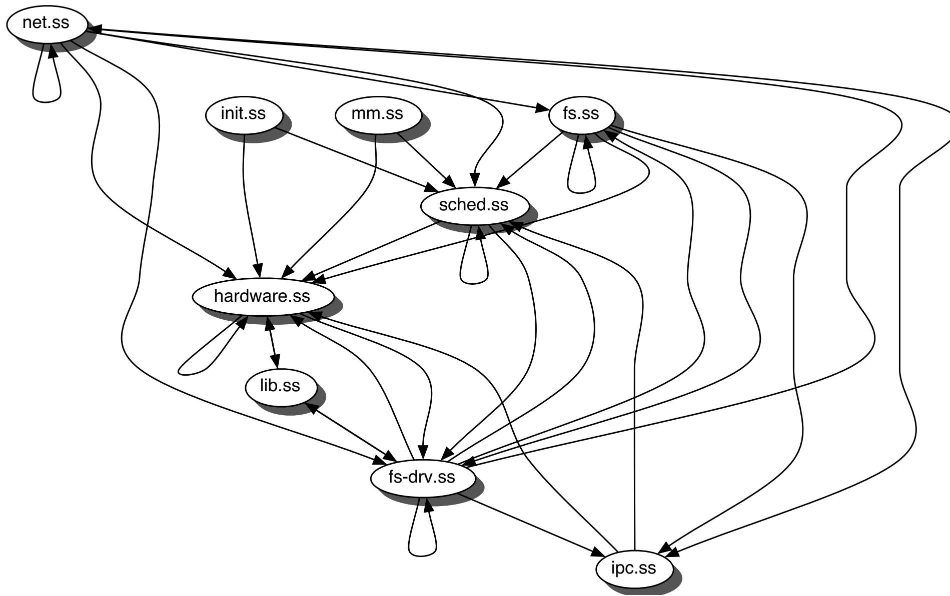


Figure 3: Lifted top-level calls—the `tlcall` graph

`contain` hierarchy. If we are interested only in the calls between a few different subsystems regardless of their position in the hierarchy, this approach will give either too much information, too little, or both. To illustrate the e xibility of our approach here, we took a slight departure from the usual transformation, and de ned the `toplevel` relation manually to consist of 9 edges between subsystems chosen at varying levels in the directory hierarchy. The resulting graph analyzed consisted of 9,059 nodes and 17,832 edges.

The analysis is speci ed in 21 lines of GENSET code as a set of three equation blocks:

```
for ss in (base contain) do
  tlcontain(ss) := (/union child in contain(ss) :
            single(child) union inherit(child));
  unmarkedChildren(ss) :=
                     contain(ss) - dom toplevel;
  inherit(ss) := (/union unmarked in
                           unmarkedChildren(ss):
                             tlcontain(unmarked));
od;

for func in (base sourcecall),
    dir in (dom tlcontain) do
  inDir(func) :=
         /union file in _funcdef(func):
                                _tlcontain(file);
  dirCalls(dir) :=
      (/union file in tlcontain(dir):
         (/union f in funcdef(file) :
             (/union g in sourcecall(f): inDir(g))
         ));
od;

for tldir in (base toplevel) do
  tlcall(tldir) := dirCalls(tldir)
                           intersect (dom toplevel);
od;
```

In the rst block, we collapse nodes corresponding to les into the nearest ancestor directory node which has been marked as (*i.e.* included in) `toplevel`: this is the `tlcontain` relation.

In the second block the `sourcecall` relation representing calls between functions is lifted to become the `dirCalls` relation, representing the presence of a call between two directories. This is computed for *all* directories, not just those marked as `toplevel`. The extraneous calls from non-`toplevel` directories are ltered out in the third block, leaving the `tlcall` relation from `toplevel` directory to `toplevel` directory.

Analysis was run on a Macintosh Quicksilver with 900 Mhz G4 and 1.2 GB of RAM. The resulting graph is given in Fig. 3. The slowest step was the 4 seconds required to read the input graph (in RSF format). After the graph was represented in memory, execution of the GENSET script completed in less than 2 seconds.

6. RELATED WORK

A variety of programmable graph transformation tools have been developed, with explicit application to problems of program and software system analysis.

The approach closest to our own is the speci cation of transformations with *relational algebra* (RA). Given the tight mathematical correspondence between binary relations and directed graphs [22], it is unsurprising that the use of RA has very natural applications in graph transformation; we have borrowed a few of the operators ourselves.

A number of programmable graph transformation systems based on extensions of this algebra have been presented in the literature [3, 4, 9, 11, 16, 17, 19]. One of the common extensions is to augment RA with a transitive closure operator (this is used in all of the works cited here). With this extension, several important graph transformations have been implemented, notably in the area of software architecture. Holt, for example, shows in [11] how to implement six important architectural transformations using his Grok system, including the lifting transformation we demonstrated in this paper.

We do not yet know how the performance of GENSET compares with that of the extended relational algebra systems across the whole range of transformations expressible in RA (with or without transitive closure). Thus far, however, our experimental results (such as the example presented in this paper) suggest that the o w equation approach embodied in GENSET will be competitive

with RA approaches. (The question of greater programming convenience is still a matter of debate.)

The primary difference between a o w equation and relational algebra approach lies in the expressive power of the underlying languages. RA by itself cannot express any kind of recursion (hence the need to add transitive closure as a magic operator). Even with transitive closure, the GENSET language is strictly more expressive than RA languages. On the other hand, we will suffer from a worse worst-case, since some queries expressible in GENSET will be of a higher complexity—indeed, since one can write divergent analyses in GENSET, our worst case is unbounded.

The other major approach to graph transformation comes from the research in *graph grammars*, particularly systems for graph-rewriting-based transformations. One of the best known of these is the PROGRES project [20, 23]. In this system, transformations are speci ed with graph-rewriting rules which are using to generate C code for a stand-alone prototype. This approach offers greater expressive power than o w equations (it is computationally complete). However, it appears to suffer from the general complexity of graph rewriting—for example, the necessity of using subgraph isomorphism detection to implement generalized pattern matching. Using PROGRES to specify several common architectural transformations, Fahmy and Holt [6] report that the system, while effective at small prototypes, is impractical for transforming the large graphs associated with real software systems.

Generic iterative equation solvers have been developed primarily within the logic programming eld. Our approach to the problem of dynamic determination of dependencies has been in uenced in part by work presented in [8]. Another generic data o w analysis tool is the Fixpoint-Analysis Machine described by Steffen *et al* [24], which handles generic instances from several classes of o w analysis whose reductions to equivalent model checking problems are known.

Recently Rayside and Kontogiannis [21] have developed another generic worklist algorithm that, like ours, is designed for towards graph-based analyses. Their work is targeted speci cally toward generic support for graph reachability problems, which leads to three signi cant differences from our version. First of all, their test for re-evaluation is speci cally for a *monotonic* change, while we base re-evaluation on a test for *any* changes, allowing for the possibility of non-monotone functions to be evaluated. Further, the user of their algorithm must specify manually the lattice to be used, and in particular, must de ne the partial order relation between elements, which is necessary for detecting monotonic change. Even if we were to enforce monotonicity, we always use the same lattice—the power set of the set of possible nodes in the relation's image, ordered by subset/superset inclusion. Finally, because their algorithm is targeted towards reachability analyses on a x ed graph, the determination of dependent equations is always done in the traditional static fashion: by taking the successors in the graph of the current node. As discussed in §3.1 above, this approach to dependency determination does not work in our more general setting.

An interesting alternative solution to the problem of representing in nite sets is Alfaro's constructive μ-calculus [5] in which GFP equations are restricted by the requirement that the universe of discourse be both nite and explicitly stated. We are still investigating a comparison of this approach with our own.

7. CONCLUSION AND FUTURE WORK

The expressiveness and relative ef cienc y of o w analysis appears to be a useful point in the design space of graph transformation tools for program analysis. Meanwhile, there remain a number of opportunities for further development.

Two aspects of the existing implementation of GENSET need improvement. First, the requirement that equation blocks be evaluated in source code order avoids the mess of mutual recursion between blocks, but is a shortcoming in the declarative character of the language. The alternative is to add a "program-wide" iteration, along with the use of dependency graphs both within and between equation blocks to determine, if possible, a better evaluation order than that given by the source code. Second, the language lacks effective schemes for parameterization and library construction. Relations that are not explicitly de ned by equations in a script are presumed to exist in a global symbol table before evaluation, with the symbol value assumed explicitly in the source code. We are currently developing a procedure-de nition facility for the reuse of commonly used equations (*e.g.* transitive closure) and will remove assumptions about the global symbols with a top-level "main" construct.

From a philosophical point of view, the paradigm of the GENSET language is compatible with relational algebra and it may bene t from the inclusion of many of the ordinary RA operators. As it stands, the language is strictly more expressive than RA, and such additions would therefore be "syntactic sugar," but may offer more convenience for programmmers.

More challenging is the possibility of developing a static type system to guarantee niteness and union-compatibility properties of relations, eliminating the need for many of the runtime checks that are performed in the present version. In addition to potential improvements in the runtime performance of interpretation, this would make the possibility of a compiler for the language more appealing.

Finally, it is important to understand the tradeoffs between expressiveness and ef cienc y among the variety of graph transformation approaches available for manipulating representations of programs and software systems. Fahmy *et al* [6, 7] have begun this task with a comparison of manipulation using relational algebra and graph rewriting. We expect that our o w analysis approach will fall somewhere between these two, but more benchmark analyses are needed with representative, practical examples. Widespread adoption of a single exchange format such as GXL may help; additionally, we are developing conversions among some of the more widely used graph representations to facilitate comparisons.

Acknowledgements
Craig Kaes, Sachiko Honda, and Lam Nguyen designed and implemented parts of GENSET. We also wish to thank Dr. Jan Hidders for a helpful discussion of relative expressiveness and Dr. Luciano Baresi for invaluable constructive feedback. Finally we thank the anonymous reviewers for their many helpful suggestions.

8. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] A. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] O. Ciupke. Analysis of object-oriented programs using graphs. In *ECOOP'97 Workshop Reader*, LNCS 1357, pages 270–271. Springer, 1998. Extended version available (as of August 2002) at `http://www.fzi.de/ecoop97ws8/`.

[4] O. Ciupke. Automatic detection of design problems in object-oriented engineering. In *Technology of Object-Oriented Languages and Systems (TOOLS 30)*, pages 18–32. IEEE Pr., 1999.

[5] L. de Alfaro. Model-checking the world wide web. In *Computer Aided Veri cation, 13th International Conference (CAV 2001)*, LNCS 2102. Springer, 2001.

[6] H. Fahmy and R. Holt. Using graph rewriting to specify software architectural transformations. In *15th Conference on Automated Aoftware Engineering (ASE 2000)*, pages 187–196. IEEE Pr., 2000.

[7] H. Fahmy, R. Holt, and J. Cordy. Wins and losses of algebraic transformations of software architectures. In *16th Conference on Automated Aoftware Engineering (ASE 2001)*, 2001.

[8] C. Fecht and H. Seidl. A faster solver for general systems of equations. *Science of Computer Programming*, 35(2–3):137–161, 1999.

[9] L. Feijs, R. Krikhaar, and R. V. Ommering. A relational approach to support software architecture analysis. *Science of Computer Programming*, 28(4):371–400, 1998.

[10] A. Geser, J. Knoop, G. Lüttgen, O. Rüthing, and B. Steffen. Non-monotone xpoint iterations to resolve second order effects. In *6th International Conference on Compiler Construction (CC '96)*, LNCS 1060, pages 106–120. Springer, 1996.

[11] R. C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *5th Working Conference on Reverse Engineering (WCRE'98)*, pages 210–219. IEEE Pr., 1998.

[12] R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a Standard Exchange Format. In *7th Working Conference on Reverse Engineering (WCRE'00)*, pages 162–171. IEEE Pr., 2000.

[13] G. A. Kildall. A uni ed approach to global program optimization. In *ACM Symposium on Principles of Programming Languages (POPL 73)*, pages 194–206. ACM Pr., 1973.

[14] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *1994 Conf. on Programming Language Design and Implemention (PLDI '94)*, pages 147–158. ACM Pr., 1994.

[15] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[16] R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, and C. Verhoef. A two-phase process for software architecture improvement. In *International Conference on Software Maintenance, Oxford UK (ICSM '99)*, pages 371–380. IEEE Pr., 1999.

[17] B. Kullbach and A. Winter. Querying as an enabling technology in software reengineering. Technical Report 22/98, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1998. `http://www.uni-koblenz.de/fb4/`.

[18] Portable Bookshelf (PBS) Tools. Available (as of August 2002) at `http://swag.uwaterloo.ca/pbs/`.

[19] A. Postma and R. Krikhaar. Applying relation partition algebra for reverse architecting. In *1999 Workshop on Software Reengineering, Bad Honnef, Germany*. `http://www.uni-koblenz.de/~ist/RWS99/`.

[20] PROGRES. Available (as of August 2002) at `http://www-i3.informatik.rwth-aachen.de/research/projects/progres/`.

[21] D. Rayside and K. Kontogiannis. A generic worklist algorithm for graph reachability problems in program analysis. In *Software Maintenance and Re-engineering, Proc. of the 6th International Conference (CSMR '02)*, pages 67–76. IEEE Pr., 2002.

[22] G. Schmidt and T. Ströhlein. *Relations and Graphs*. Springer-Verlag, 1993.

[23] A. Schürr, A. Winter, and A. Zündorf. PROGRES: Language and environment. In G. Rozenberg, editor, *Handbook on Graph Grammars: Applications, Vol. 2*, pages 487–550. World Scienti c, 1999.

[24] B. Steffen, A. Claßen, M. Klein, J. Knoop, and T. Margaria. The xpoint-analysis machine. In *Concurrency Theory, 6th International Conference (CONCUR '95)*, LNCS 962. Springer, 1995.

[25] SUIF. Available (as of August 2002) at `http://suif.stanford.edu/`.

Refining Code-Design Mapping with Flow Analysis [*]

Xiaofang Zhang
University of Oregon
Dept. of Computer &
Information Science
Eugene, OR 97403-1202
xzhang@cs.uoregon.edu

Michal Young
University of Oregon
Dept. of Computer &
Information Science
Eugene, OR 97403-1202
michal@cs.uoregon.edu

John H. E. F. Lasseter
University of Oregon
Dept. of Computer &
Information Science
Eugene, OR 97403-1202
johnfl@cs.uoregon.edu

ABSTRACT

We address the problem of refining and completing a partially specified high-level design model and a partially-defined mapping from source code to design model. This is related but not identical to tasks that have been automated with a variety of reverse engineering tools to support software modification tasks. We posited that set-based flow analysis algorithms would provide a convenient and powerful basis for refining an initial rough model and partial mapping, and in particular that the ability to compute fixed points of set equations would be useful in propagating constraints on the relations among the model, the mapping, and facts extracted from the implementation. Here we report our experience applying this approach to a modest but realistic example problem. We were successful in expressing a variety of useful transformations very succinctly as flow equations, and the propagation of recursively-defined constraints was indeed useful in refining the mapping from implementation to model. On the other hand, our experience highlights remaining challenges to make this an attractive approach for general use. Special measures are required to identify and remove inconsistent constraints before they propagate through a system. Also, while the required flow equations are succinct, they are also rather opaque; it is not obvious how their expressive power might be preserved in a more accessible notation.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement —*Restructuring, reverse engineering, and reengineering*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms,Design,Languages

Keywords

flow analysis, constraint propagation, architecture recovery

1. INTRODUCTION

When faced with a system re-structuring task and only a partial understanding of a system, an engineer needs both to refine an understanding of the overall system structure and to understand how individual parts fit in the whole. There exist a variety of reverse engineering techniques that attempt to infer overall system structure from detailed structure of an implementation, and also techniques for checking for conformance between a posited high-level structure and the detailed structure of an implementation. We address a related but somewhat different problem of completing and refining both an initial rough design model and an initially partially-defined mapping from implementation entities and relationships to the model.

The specification of equational constraints in the style of flow analysis is appealing both for the compactness of the equations and because simple, efficient solution algorithms are available. We posited that such specifications could be useful in refining a design model and the mapping from implementation structure to model. We had previously used a flow analysis toolkit to perform "lifting" transformations to extract a high-level model from implementation-level relations in the Linux kernel [10], duplicating reverse engineering operations that others have achieved using relational algebra [7, 13] and other means. We expected that fixed-point set equations would provide additional power to use constraints on connection structure in refining the mapping between implementation-level "facts" extracted from source code and a high-level model.

In this paper we describe experience applying this approach in a small but realistic exercise. The *GenSet* tool-kit for set-based flow analysis has been constructed piecemeal over a period of years by several developers, most of whom are no longer present. Our situation was much like that described by Murphy et al [15]: We were able to sketch a rough and inexact architectural model of the system and

[*]Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0620 and F30602-97-2-0034. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for pro t or commercial advantage and that copies bear this notice and the full citation on the rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior speci c permission and/or a fee.
SIGSOFT'04/FSE-12, Oct. 31–Nov. 6, 2004, Newport Beach, CA, USA.
Copyright 2004 ACM 1-58113-855-5/04/0010 ...$5.00.

of its relation to implementation entities and relations, but lacked the confidence and detail required to make desired changes. The first author, who was not among the developers of *GenSet*, set out to refine the model using *GenSet* itself. Implementation relations (*facts* in the terminology of the reverse engineering literature) were extracted from Java byte code (.class files), and several manipulations of those facts and the initial rough model were programmed in the *GenSet* little language of flow equations. The initial model, partial mapping, and flow equations were iteratively refined until they produced a full mapping of implementation structures to model.

The primary novelty in this process, and the chief contribution of this paper, lies in the way a partial mapping of implementation structure to high-level model is extended to a more complete mapping by solving a system of simultaneous set equations. Although the equational specifications resemble those used in data flow analysis, they do not model the flow of data during program execution. Rather, these "flow equations" describe constraints imposed on the mapping by the high-level model, and the solution at a node in the implementation structure represents the set of model entities to which an implementation entity may be mapped, consistent with the model structure. This contrasts with prior work in reverse engineering, where the mapping of implementation entities to design entities is a distinct step providing input to the process of extracting high level structure [13] or checking conformance [15].

Our experience applying this technique was positive in that we were able to refine an initial model and extend an initial mapping quite effectively. The flow equations used in this process, most of which could be re-used in other applications, are quite succinct, and the actual process of writing and refining the model and writing scripts to express our equations was completed in a few hours. Balanced against this encouraging result are some obstacles, not all of which have been satisfactorily overcome. The effectiveness of flow analysis in propagating constraints becomes a liability when mistakes in the initial model or mapping, or violations of structuring rules in the implementation, produces an inconsistent (over-constrained) system. Moreover, while flow equations can specify this constraint propagation and a variety of other useful manipulations succinctly, they can be rather opaque; it is not obvious to us how to preserve their expressive power in a more accessible notation.

2. APPROACH

The observation on which our analysis rests is simple: If we are given a partial assignment of implementation entities to design entities (allowing many implementation entities to be grouped in a single design entity), then this mapping imposes certain constraints on the remaining (unmapped) entities. Although there may be several choices for completing the assignment, only some of them are consistent with the design; other ways of completing the assignment can be ruled out. For example, if the design does not permit calls from implementation entities in design entity L1 to implementation entities in design entity L2, and if implementation entity A is assigned to L1 and makes a call on entity B, then assigning B to entity L2 would be inconsistent with the design.

Multiple constraints from a design model can be combined and propagated to extend the mapping by finding additional (initially unmapped) implementation entities that must be mapped to particular model elements in order to satisfy the constraints. Consider the example in Figure 1. Here, the "uses" relation among design entities indicates that implementation entities assigned to L1 and implementation entities assigned to L2 are each permitted to make calls on implementation entities assigned to L3, but no such dependence between L1 and L2 is permitted. No restriction is placed on calls between implementation entities assigned to the same design entity. Implementation entities A, E, and D have been assigned to L1, L2, and L3, respectively, by some unspecified method, but no assignment has been given for B and C. A and E call B, B calls C, and C calls D; these calls should be consistent with the "uses" relation in the design model. Given this information, we can infer that the only consistent assignment of B and C is to L3. Thus, the initial mapping of implementation modules A, E, and D to design entities can be automatically extended with mappings of implementation entities B and C.

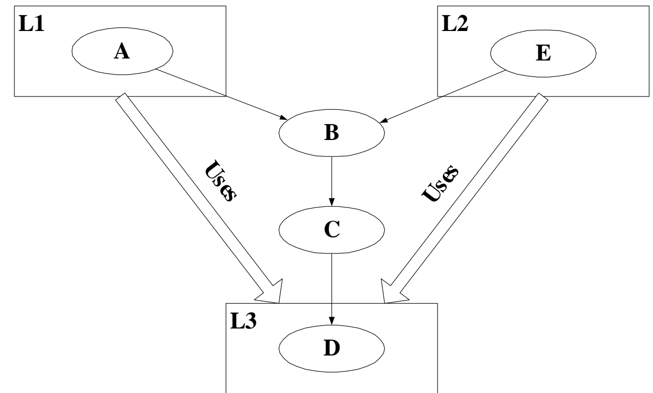


Figure 1: A partially-completed mapping from implementation-level entities A–D to high-level design entities L1–L3. If the dependence among implementation entities must conform to the dependence (uses) relation among design entities, then the only consistent completion of the mapping assigns B and C to L3.

Our analysis is designed to automate this inference, starting from a rough design model and facts extracted from an implementation, and propagating constraints to identify implementation entities whose place in the design can be determined. Since the initial design model is unlikely to be either complete or perfect, and since the implementation may violate some constraints imposed by the design, it is unlikely that the design mapping will be completed in a single analysis step. Rather, refining a model and mapping is an iterative, partially automated process in which inconsistencies are discovered and removed and some under-constrained implementation entities are manually assigned to design entities.

2.1 Initial model and mapping

We assume that engineers can provide at least a rough model of the overall system. We also assume that each implementation level entity is associated with ("belongs to") a single entity in the design model, and that an engineer can provide at least a partial (albeit incomplete) version of

this mapping. Both the gross structure and the association of implementation to model are represented as relations between entities. In general we expect that a relation among implementation entities (e.g., "is a subclass of") is permitted only if there is a corresponding relation (e.g., "depends on") among the model entities that is consistent with the implementation-to-design mapping.

The correctness of our approach does not depend on a particular kind of high-level model. It does require that design entities (which we will henceforth call "modules") can be viewed as aggregations of implementation entities, and that relationships between two modules can be interpreted as allowing corresponding relationships among their elements, without constraining relationships among elements of the same module. Relationships between modules must be asymmetric to provide useful constraints. Hierarchical relations, such as dependence in layered designs, have the required properties. While our approach can easily be extended to use several different relations among modules, corresponding to different relations among implementation entities, for simplicity here we consider a single dependence relation. This is consistent, for example, with the kind of high level model used by Murphy et al [15].

The initial system facts (input to our analysis) describe two sets of entities, MOD and IMP, which represent, respectively, the modules in the high-level model and the entities of the implementation level. They are:

- $use_I \subseteq IMP \times IMP$ — a relation modeling dependence between entities in IMP.
- $use_M \subseteq MOD \times MOD$ — a dependence relation on the modules in MOD.
- $belongs \subseteq IMP \times MOD$ — a partial mapping of implementation entities in IMP to modules in the high-level model.

There are a number of ways these facts could be obtained. The use_I relation can be inferred from an extracted call graph, object references in the byte code, def/use chains, or any number of other ways. Reverse engineering tools such as Rigi [14] are available for such tasks, along with standard cross-reference or profiler tools, simple scripts, bytecode engineering tools (*e.g.*, Apache BCEL[4]), and so on. A first approximation of the high-level model (from which the use_M relation comes) may be available in system documentation, or can be supplied by a knowledgeable engineer [9]. The *belongs* relation, an assignment of a few implementation entities to seed the analysis, can be reasonable guesses with or without the aid of reverse engineering tools.

2.2 Analysis

Relations representing the initial facts of the system induce constraints on the way entities can be mapped in the model. We can propagate these constraints to extend the mapping by finding additional entities that can only satisfy the constraints if they are mapped to particular model entities. While we assume that the intended high-level design associates each entity with exactly one module, we cannot expect an exact inference of this association from initial facts that are incomplete, particularly if there are inaccuracies in the design model or inconsistencies between design and implementation. Each time we run the analysis, we obtain not a complete mapping of implementation entities to design modules, but a mapping of entities to *sets* of modules to which they could be mapped, given the current set of facts.

The result of the analysis is a new binary relation, *mbt* ("modules belonged to"), which defines for each implementation-level entity x an association with a set of modules in MOD, $mbt(x)$.[1] The constraints solved by the analysis express the requirement that for every entity x, we have $m \in mbt(x)$ only if m is a module to which x may be mapped in a manner consistent with the rest of the implementation-level and high-level structure.

To aid in the definition of *mbt*, we also introduce new auxiliary relations $mu, mub \subseteq IMP \times 2^{MOD}$. The first of these ("modules used") associates with each x in IMP the set of modules on which x depends, while the latter ("modules used by") gives the modules containing implementation-level entities that use the modules to which x might belong. The relation *mbt* is then defined to be the largest relation that satisfies for all $x \in IMP$ the following system of simultaneous set equations:

$$mu(x) = \begin{cases} \bigcup_{m \in mbt(x)} use_M(m), & \text{if } belongs(x) = \emptyset \\ \bigcup_{m \in belongs(x)} use_M(m), & \text{otherwise} \end{cases}$$

$$mub(x) = \begin{cases} \bigcup_{m \in mbt(x)} use_M^{-1}(m), & \text{if } belongs(x) = \emptyset \\ \bigcup_{m \in belongs(x)} use_M^{-1}(m), & \text{otherwise} \end{cases}$$

$$mbt(x) = \left(\bigcap_{w \in use_I^{-1}(x)} (mbt(w) \cup mu(w)) \right) \cap \left(\bigcap_{y \in use_I(x)} (mbt(y) \cup mub(y)) \right)$$

The intuition here is essentially the one given above: in order to be a candidate module for mapping of x, the structural context of $mbt(x)$ in the model (*i.e.*, in the use_I relation) must be consistent with the structural context of x. We assume, although we cannot guarantee, that the partial mappings already supplied by the user enjoy this property. The user-supplied partial mapping *belongs* overrides the computed relation *mbt*.

The desired solution of *mbt* is the greatest fixed point of the constraint equation, while both *mu* and *mub* are least fixed points. Evaluation of all three equations is easily seen to be monotone, so these solutions exist and are computable by iteration.

To illustrate the constraint propagation, consider again the example in Figure 1. Here the given assignments of A to $L1$, E to $L2$, and D to $L3$ are the initial *belongs* relation. The use_I edges (A, B) and (E, B) constrain $mbt(B)$ to the set of modules used by both A and E. Similarly, the edge from B to C constrains $mbt(C)$ to the set of modules used by $mbt(B)$. Combining constraints on nodes B and C gives the singleton sets $mbt(B) = mbt(C) = \{L3\}$.

[1] For a binary relation R, we let $R(x)$ denote the set $\{y \mid (x, y) \in R\}$, while the "inverse" $R^{-1}(x)$ denotes the set $\{w \mid (w, x) \in R\}$.

2.3 Using the results

In the ideal case, the user has made no mistakes in the initial model and has provided just enough of the partial implementation-to-module mapping to guarantee that analysis will map each unmapped entity to a unique module. If this is the case, then the mapping of implementation-level entities to modules is complete upon convergence of the constraint analysis, making our process fully automatic. Of course, this will almost never happen. Reliance on the initial model is unavoidable, but in doing so, we are vulnerable to the companion threats of incomplete and/or inconsistent information.

Put another way, after propagation has converged, there are three possible results for each node of the source model: it may be assigned to a single module, or a set of different modules, or the empty set. These possibilities correspond to the ideal or exactly-constrained case, under-constrained case and over-constrained case, respectively. We will explain the latter two in the remainder of this section, discussing ways to use the analysis results in refining both the model and mapping. In each case, we get *some* information about our high level model and the original assignment. We can use this information to modify the high level model or change the mapping and repeat the process until we get a consistent view of the high level model and a complete mapping to the source model.

2.3.1 Under-constraint

When the constraints on an entity produced by the initial system facts are not strong enough to narrow the mapping of the node to a single module, we say that the entity is *under-constrained*. There are a number of ways that could lead to this situation. Here we will describe two common cases.

The first case is an imprecise high level model. Since usually our understanding of the high level model is approximate and incomplete, it is very possible to have superfluous use_M edges in the model.

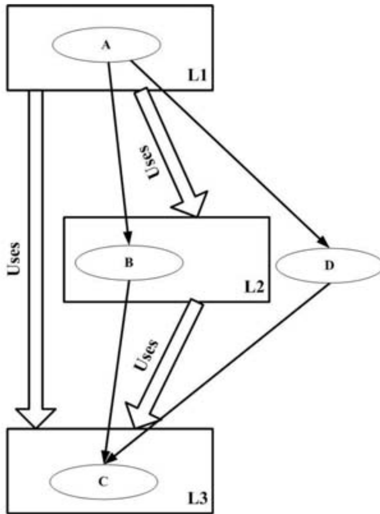


Figure 2: An example of under-constraint. D **can be mapped to any of the three modules.**

Consider the example shown in Figure 2. If the $uses_M$ edge $(L1, L3)$ is not present, the constraint propagation will map D to the set $\{L1, L2\} \bigcap \{L2, L3\} = \{L2\}$. But with the presence of $(L1, L3)$, D is mapped to the set $\{L1, L2, L3\}$. Whether this instance of under-constraint is a flaw in the high level model depends on the intent of the designer. A high level model whose dependence relation is not sufficiently sparse to provide useful constraints might suggest a need for refactoring.

A second cause of under-constraint arises from leaf nodes in the source model, a situation that is illustrated in Figure 3.

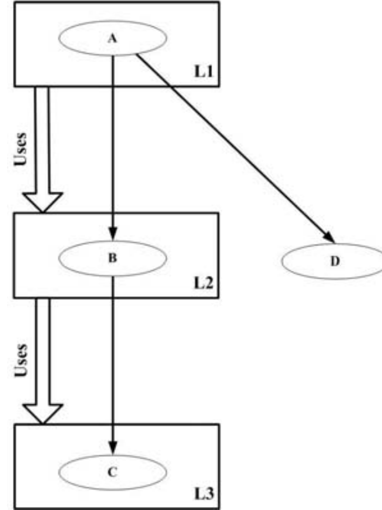


Figure 3: Another example of under-constraint. D **can be mapped to either** $L1$ **or** $L2$**, although in many cases it will correspond to a helper component for** A **in the implementation.**

A leaf node corresponds to an implementation-level entity that does not use other, distinct entities of the implementation. These are often helper functions, helper classes, etc. In most instances, they should probably belong to the same module as the entities using them, but this is a heuristic and not a part of the analysis itself. Consequently, these entities are often under-constrained. Although at times useful, incorporating such a heuristic in the analysis would not be correct in all situations. We prefer to show these nodes to the engineer, leaving the choice to her.

2.3.2 Over-constraint

When the constraints on a source node produced by the original assignment and the subsequent propagation conflict with each other, then we say the entity is *over-constrained*. Usually, an over-constrained entity will be assigned to an empty set of modules. A simple example is given in Figure 4.

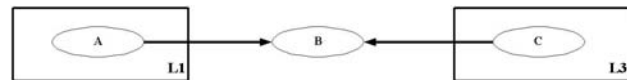


Figure 4: An example of over-constraint. B **cannot be mapped consistently to either module.**

The consequences of over-constraint are more serious than under-constraint. Where under-constraint tends to result in mappings that require manual intervention to complete, the

rapid propagation of over-constraint may render the whole analysis useless. An over-constrained entity will tend to over-constrain all adjacent entities; over-constraint spreads like an infection.

As with under-constraint, an imprecise high level model is a possible culprit. To consider again the example in Fig. 4: there is no use_M edge between module $L1$ and $L3$, and hence B cannot be mapped to either module. A system in which no module uses another is highly suspect, so in this case, the solution might be to modify the high-level model and add the edge $(L1, L3)$ to the use_M relation. In practice, however, this sort of flaw in the model may be hard to detect from the analysis result, since a mapping to $\emptyset$ cannot give us much information for diagnosis.

As a debugging aid, we can use a modified form of the analysis in which we explicitly ignore an entity as soon as we find that it is over-constrained. One way to do this is to restrict the equation for mbt so that it is defined only on entities in the domain or codomain of the use_I relation that use or are used by entities already mapped to modules:

$$\begin{aligned}
mu(x) &= (\ldots) \\
mub(x) &= (\ldots) \\
R(x) &= use_I(x) \setminus \{x' \mid mbt(x') = \emptyset\} \\
_R(x) &= use_I^{-1}(x) \setminus \{x'' \mid mbt^{-1}(x'') = \emptyset\} \\
mbt(x) &= \left(\bigcap_{w \in _R(x)} (mbt(w) \;\cup\; mu(w)) \right) \\
&\quad \bigcap \\
&\quad \left(\bigcap_{y \in R(x)} (mbt(y) \;\cup\; mub(y)) \right)
\end{aligned}$$

However, this is only a heuristic for debugging and not a part of the modules analysis per se, as it destroys desirable properties of the constraint equations, especially monotonicity. Propagation will still terminate — only a finite number of entities can be thrown out — but the final result for mbt is not necessarily deterministic. In particular, by aborting evaluation on certain elements, we do not get a fixed point for mbt, and hence the "solution" does not satisfy the constraints for all entities in IMP. What this does give is the possibility of seeing "snapshots" of the propagation, which become frozen as the domain of mbt is truncated. Results will likely resemble under-constraint cases, and could perhaps be used by the designer to isolate the fault within the initial system facts.

2.3.3 *Practical complications from cycles*

If a layered architecture or other hierarchical design structure is used, there should be no circular dependencies in the high-level model. However, circular use is fairly common in the implementation; references in object-oriented programs, for example, especially in dynamic-linking languages such as Java. Further, the use_I relation may itself abstract lower-level dependencies, which can induce cycles in the relation. For instance, an implementation entity (e.g., a Java `package`) can contain multiple classes, and if we are extracting implementation dependency from class references and method calls, cycles are still possible at the package level even if there is no mutual reference at class level. Therefore we have to face the mismatch between the acyclic high-level model and the possibly cyclic low-level source model.

The effect of cycles can be subtle. As illustrated in Figure 5, the mismatch between the acyclic model and cyclic implementation dependency can result in both under-constraint and over-constraint.

In the left side of Figure 5, B will be under-constrained and our analysis will map B to $\{L1, L2\}$. If the direction of the use_M edge between $L1$ and $L2$ is inverted, as in the right side of Figure 5), then B will be over-constrained and our analysis cannot map it to any module. In either case, cycles may compromise the usefulness of the analysis results.

On the other hand, low-level cycles are not necessarily troublesome. Most cycles reside inside only one module and will not violate the acyclic nature of the high-level model. This case does not complicate the constraint propagation in our analysis. If one initially maps just one entity on the cycle to one module, then every other node on the cycle will be absorbed into the same module.

Therefore, we need only to be careful with cross-module cycles. However, this begs the question: If the mapping from the low-level source model to high-level moduleing model is not complete, how do we know that a cycle is a cross-module cycle? Our approach is the following: try to find all low-level cycles first and then try to break those that are likely cross-module cycles.[2] For the other cycles, assume that all entities on the cycle belong to just one module.

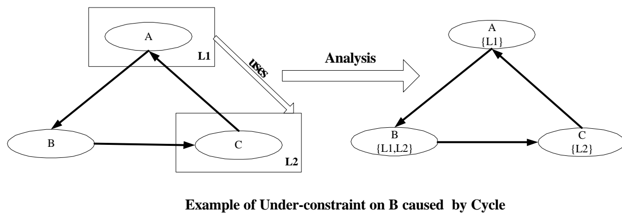
It is worth noting that certain cross-module cycles do not really exist in the implementation-level dependencies, but are an artifact of imprecision in construction of the use_I relation. This is particularly true when this relation abstracts multiple lower-level dependencies. For instance, an upcall may introduce a use_I edge from a lower-module entity to higher-module one, and in turn induce a cross-module cycle. In many cases, however, the lower module can run correctly without the callee at higher module in the upcall. In such cases, the use_I edge corresponding to the upcall is not a true dependency edge, and therefore a cycle caused by this edge is not a true dependency cycle.

By excluding the upcall edge from our analysis, we can eliminate this kind of cycle. However, the general problem of detecting false alarms in the cyclic dependencies is a bit harder. Harmless upcalls occur frequently in the form of callbacks, but many design idioms other than callback can also produce false dependency cycles.

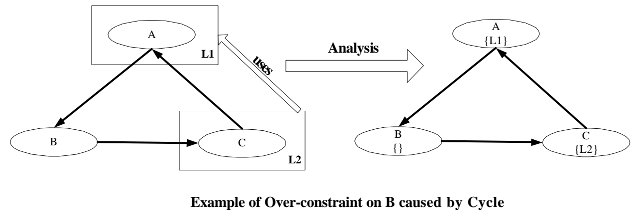
In this paper, we will not try to provide a general and precise solution, instead focusing only on callback detection. For this project, we use an implementation-level pattern to detect callback cycles, although this is complicated by the fact that there is no common, formal definition of callback. Which function call is a callback seems to be heavily dependent on the programmer's mental model. Further, there are many ways to implement callbacks in source code.

The implementation-level pattern illustrated by Figure 6 is one that is frequently used by programmers. In this pattern, class B implements an interface for the callback methods. Class A calls these methods indirectly through interface I. Because of its flexibility, this pattern is often used in Java programming, including the event-listener mechanism in Swing, the visitor pattern, etc. While useful, this pattern,

[2] By "breaking" a cycle, we mean that one or more of the cycle edges is removed from the set of initial system facts.



Example of Under-constraint on B caused by Cycle



Example of Over-constraint on B caused by Cycle

Figure 5: Potential effects of cycles in the implementation-level dependencies.

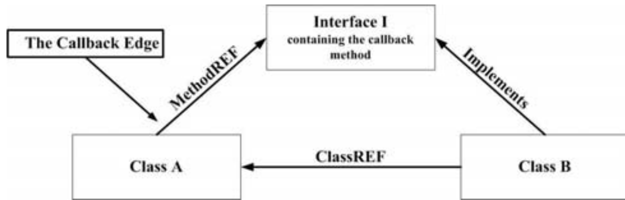


Figure 6: A pattern common to many upcalls, including callbacks

as with other heuristic devices, is neither necessary nor sufficient. We still have to resort to programmers for the final decision of whether it is a callback.

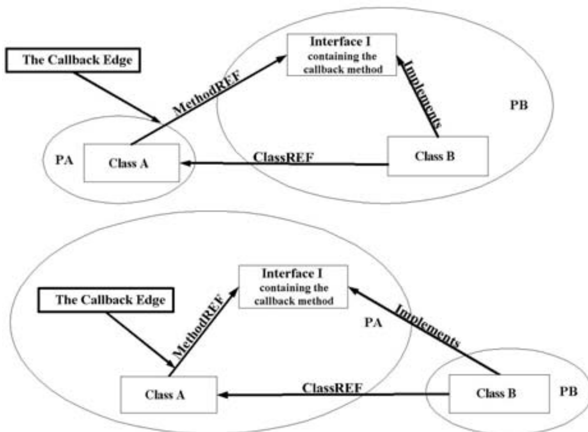


Figure 7: Call-back and cycle: In the lower diagram, there is no reference cycle at all.

Finally, we note that low-level upcalls do not necessarily introduce implementation-level dependence cycles at all. As shown in Figure 7, if the interface I and class B belong to package PB and if class A belongs to package PA, then there will be a cycle of reference between these two packages; but if I belongs instead to package PA, then there will be no cycle of reference between these two packages.

Eventually, after the detection and breaking of spurious cycles, some real cycles may remain. If these cycles cross module boundaries, they cannot conform to an acyclic "uses" relation among modules. Diagnosis of the violation may require inspection of relevant source code, and repair requires restructuring it.

3. EXPERIENCE

To evaluate the approach explained in Section 2, we implemented the analysis using our *GenSet* tool, and then applied the analysis to the *GenSet* software itself. *GenSet* [10] is a generic programmable tool for flow equation-based transformation of graph-structured data, driven by a "little language" for transforming attributed digraphs.

Although, with just 196 Java classes, *GenSet* is comparatively small, the exercise was realistic in that the subject system was developed over a period of years, and most of the original developers are no longer available to us. As with many such projects, there has been considerable elaboration and deviation from the original system structure. We have an agenda of desirable modifications (e.g., a batch interface that is not part of the graphical user interface), but the original design documents can no longer be trusted to guide maintenance. Applying the analysis to *GenSet* itself was therefore attractive as it fit our assumptions of a system in which there is an initial, imperfect design model, and where at least a few initial assignments of implementation entities to high level model could be made. Knowing that our approach was unlikely to work perfectly also made a small and fairly familiar example attractive.

3.1 Steps in analysis

Refining and elaborating the design model and code-to-design mapping is an iterative, semi-automated process, beginning with (manual) construction of an initial design model and (automated) extraction of relations ("facts") from the implementation. In each iteration, we applied the automated analysis in attempt to complete the mapping. Initially the mapping resulted in some code entities being under-constrained, while others were over-constrained and could not be mapped to any design entity. After diagnosis of these faults, the model and initial mapping were modified, and the process was repeated.

In addition to implementations of the mu, mub, and mbt equations, a number of *GenSet* scripts were used to diagnose the problems. Although these auxiliary scripts form an important part of the refinement process, they are peripheral to the basic technique, and space considerations prevent their inclusion in this paper. For the interested reader, they are available as an electronic appendix at [19].

An initial design model of the *GenSet* system was created by one of the original developers from his working knowledge of the system, together with some inferences from the directory structure of parts constructed by others. This model turned out to be nearly correct. Relations among Java classes and packages were extracted from Java byte code; class-level relations were "lifted" to package-level re-

lations, following a transformation described in [10].

Since dependence in our design model is acyclic, we know that cyclic reference among packages is likely to cause over-constraint. Rather than wait to detect and debug the resulting unassignable nodes, we began with a *GenSet* script to detect all reference cycles in the (lifted) source code model. Where genuine cycles (*i.e.*, those not resulting from harmless upcalls) were present, we assigned the entities in a cycle to a single model entity.

With both the high-level design model and the low-level source code model, we then proceeded to relate these two models by assigning some low-level nodes to the high-level nodes. The cues in the source code such as file or component names, directory structure and so on enabled us to make this first guess.

This first assignment produces constraints on the nodes that are related to the assigned nodes. After analysis to propagate those constraints, we may still have under-constrained nodes and over-constrained nodes. Then we can locate these nodes and diagnose problems by comparing the corresponding part of the high-level model and the source code model. Depending on the causes, we can either modify the high-level model or the assignment of low-level model nodes.

Since it is usually not possible to find all flaws in the assignment or in the high-level model in just one round, we may need to repeat the steps above several times (not including cycle detections). In the *GenSet* example, three rounds were enough for us to get a satisfactory mapping.

In the rest of this section, we will describe in turn the high-level model, the source code model, the cycle detection and how we refined the mapping in the *GenSet* example.

3.2 The high-level model

As shown in Fig 8, the high-level model we posited for *GenSet* consists of 6 layers.

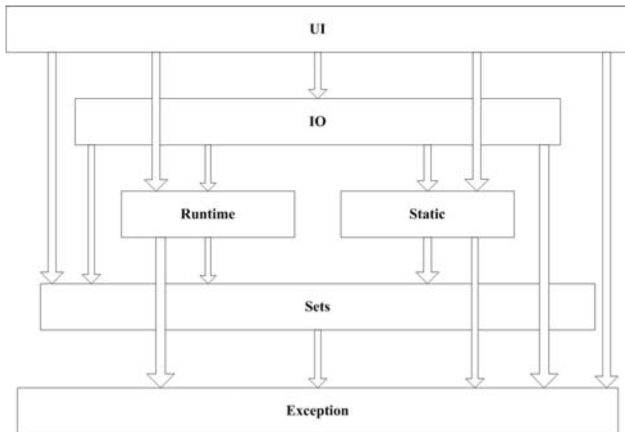


Figure 8: The high level model of the *GenSet* system

The UI layer interprets user commands and drives the *GenSet* engine. The IO layer is in charge of reading digraph data and *GenSet* scripts and writing results. The results may be presented as text or piped to dotty [12] for graphical presentation as a directed graph. The Runtime layer, Static layer and Sets layer constitute the engine of *GenSet*. The Static layer includes parsing and static semantic analysis of scripts. The Runtime layer executes the scripts, producing output data that is forwarded to the IO layer for output and the UI layer for display. Set is the core data structure module used by the Static layer and Runtime layer. All *GenSet*-specific exceptions, including run time and (script) compile time exceptions are grouped in their own layer. These layers are intended to have an acyclic "uses" relation, following established principles for maintainable systems [16], and references among entities in these layers should be consistent with the "uses" relation.

3.3 The source code model

In our source code model the elements of IMP are Java packages, and the relation use_I is derived from references from classes in one package to classes in another package.

Since *GenSet* is implemented entirely in Java, it was straightforward to extract references from byte code (.class) files using a disassembler (Apache BCEL) and simple scripts for textual pattern matching. Field references, method references, inheritance from super-classes and implementation of interfaces were modeled; references to packages outside *GenSet*, like java.lang.*, were excluded. A *GenSet* script was used initially to lift all kinds of references to a single "REF" relation among Java packages; another relation "contains" represented package nesting. The remainder of the analysis was performed at the level of packages. After eliminating self-references, the IMP relation of references between packages was a graph of 24 nodes and 82 edges.

3.4 Cycle detection

Since the "uses" relation of the design model is acyclic, we know that cycles in references among packages are likely to lead to over-constraint. The core layers analysis algorithm effectively detects over-constraint but is not very useful for diagnosing it and suggesting repair, so we prefer to detect and treat cycles in a preparatory step. We wrote a very simple *GenSet* script which essentially computes the transitive closure of references from each node. Edges that lie on cycles become part of a new relation "sliced_packageRef," shown in Fig 9.

In Fig 9, the left strongly connected subgraph includes many small cycles. From the names of the packages, one can guess that all the packages in this subgraph should belong to the UI layer. Cycles within layers do not violate the design model, so we need not be concerned with this one. If we assign any source component from this cycle to the UI layer, the rest will automatically be assigned to the same layer by the layers analysis script.

The other strong-connected subgraph is not so easily disposed of, since it is not clear to which layer each package should belong. We first guessed that the cycle of references was an artifact of call-backs, through which a "calls" reference might be reversed with respect to a "uses" relation between layers. Another simple *GenSet* script was written to detect call-backs in the pattern shown in Figure 6. This script found 17 callback edges, almost all of which occur in an instance of the visitor pattern. Contrary to our guess, however, none of the call-backs was responsible for the cycles between packages: all of them had been implemented in the manner depicted by the lower diagram of Figure 7. We therefore concluded that this cycle represented a real violation of the design model.

We used another *GenSet* script to "drill down" and view the corresponding references among classes, shown in Fig-

Figure 9: Cycles in *GenSet*

ure 10. We can see that if we move `DASet` to another package (we chose `Set`), then the cycle between `Semant.static` and `Environment` can be eliminated. Re-running the cycle detection script on the modified structure (modifying the fact base as a kind of "what if" analysis of proposed change to the implementation) showed that this change would not introduce any new cycles.

It was not so easy to see how the cycle between `Tree` and `Semant.Static` should be broken. Rather than resorting to detailed reading of source code, we elected to leave this cycle initially.

Figure 10: Drill down

3.5 Refining the mapping

After dealing with cycles, we manually assigned six nodes, one to each layer, using package names as a hint. The layering analysis script was then invoked to assign the other 18 nodes automatically. The mapping of most nodes was narrowed to one or two candidate layers. However, five leaf nodes were completely unconstrained, associated potentially with every layer, and two nodes were over-constrained (associated with no layers). We manually assigned the five completely unconstrained nodes to the layer belonging to their immediate parent, and re-ran the layers analysis. All nodes except the two over-constrained ones were then assigned to a single layer.

One of the two remaining nodes is `Semant.Worklist`. To diagnose the problem, we used another *GenSet* script to focus on it by extracting a subgraph containing just this node and edges to and from it. As shown in Fig 11, since there is no edge from the Runtime layer to the Static layer, `Semant.Worklist` can not be assigned to either of these two layers. We concluded that the original high-level model was not complete, and added an edge from layer Runtime to layer Static.

After this minor adjustment to the high-level model, rerunning layers analysis assigns all nodes, and in almost every case, each to a single layer; one node is marked as potentially being mapped to either of two layers.

Figure 11: over-constrained node

4. SUMMARY AND DISCUSSION

After completion of the analysis, we have studied some of the source code where we had most uncertainty about the results, and have found them satisfactory. (We cannot really say "correct," since there may be many valid ways to organize and represent the same set of implementation components.) It is reassuring that we did find one actual flaw in the design model (the missing dependence of the Runtime layer on the Static layer), in addition to one actual design violation in the source code (the cycle between packages Semant.Static and Tree).

The *GenSet* scripts and the approach for using them were developed over a period of weeks. To determine a lower bound on how long the exercise might take if all the scripts were already available at the outset and the general strategy for using them well-understood, we repeated the process from beginning to end, including extraction of facts from source code, computation and interactive display of each intermediate result, and all manual steps in creating and refining the model and mapping. This took somewhat less than two hours, and even if the approach were very routine this does not fully account for "think time" to understand intermediate results. On the other hand, mature tools could make the process less cumbersome than applying a generic tool like *GenSet* as we have done here.

Since engineers iteratively specify, compute and refine the code-design mapping, the performance and scalability of our

approach depends on both the algorithmic part and the required human reasoning. Happily, the constraint-based approach offers a strong separation of concerns between the two parts. With our choice to express all constraints in the style of flow equations, we can use for a solver simple algorithms of known effectiveness, and we have previously applied the flow analysis tool to more substantial examples including extraction of a high-level model from the Linux kernel, which required only a few seconds [10].

Our primary concern, then, is not performance and scalability of the tool, but rather whether the demand on human involvement can be prevented from growing excessively on much larger problems. Ultimately, while we might have read the source code involved in each of the over-constrained subproblems in our small example, that would not be feasible in refining the design model of a large system. We have therefore focused on devising approaches in which a single pattern for resolving a problem in mapping (e.g., recognizing call-backs) can be applied to many instances of the problem. Experience with other and more complex systems is needed to determine whether such patterns can generally be developed with modest effort, and whether a few patterns are widely applicable.

5. RELATED WORK

The work described here is related to prior work in software system comprehension and reverse engineering (the problem domain we address) and to work in the application of set-valued constraint solvers (the technique we apply). Set constraints have previously been applied to express a variety of program analyses, including type inference, closure analysis, and classical data flow analysis [1]. Since we restrict our set constraints to the same form as data flow equations, we are able to use a version of the well-known workset solution algorithm [10], which in practice is more efficient than the standard chaotic iteration and resolution approaches necessary for more general constraint forms.

The completion of the high-level model in our approach resembles the basic task involved in constructing software reflexion models [15]. Such models summarize the structural information collected from the implementation, in the context of a high-level model provided by the engineer. From this mapping, a comparison between high-level and implementation level structure can be made: the user can view high-level edges that agree with the implementation structure, edges in the implementation that are missing from the high-level model, and edges in the high-level model that have no corresponding edge in the implementation. However, in the form described by Murphy et al, the mapping between implementation and high-level structure must be completely specified by the engineer. If a partial mapping is provided, the implementation-level components not mapped are simply elided from the final model. The automation provided by our constraint propagation should prove a useful aid to this process.

Like our work, Sartipi and Kontogiannis [18] offer a top-down approach to the model-completion problem, in the sense that the architecture recovery process starts from a rough high level model, which is iteratively refined by automatic analysis and user intervention. Their approach also relies on an attributed digraph model of system and implementation and on a constraint-based specification. However, their constraints are mainly numerical ranges to restrict what can be mapped to high level entities. This is designed to facilitate the computation of "closest" solutions, in terms of empirically-derived metrics, which can provide useful information to the user in the case of over or under-constraint. Their constraint solving is based on graph pattern matching, the intractability of which has frustrated previous efforts [6]. Aware of this issue, the authors (with some success) address it heuristically by breaking a problem into smaller parts and computing sub-optimal matches for each part.

The Rigi system [14] is a well-known tool for subsystem identification and composition, implementing a number of semi-automated techniques. Identification of potential abstraction in the system is based on various software metrics, which can be controlled by the engineer. The tool supports interactive construction and refinement of the high-level model, so that the engineer can determine by experiment the most appropriate structure.

Finally, a number of authors have developed approaches for automatic or semi-automatic recovery of high-level structure based on the use of relational algebra. Work along this line generally relies either on a domain-specific language to express queries in a binary relational algebra extended with a transitive closure operator [7, 8, 13] or on a deductive database language [3]. Others have implemented similar queries in logic programming languages [2, 5]. Our use of a purely relational model of system facts and the manipulations we apply are closely related, but rather than adding transitive closure to relational algebra, we obtain more expressive power (without surrendering decidability) by using iteration to least and greatest fixed points. This was crucial for finding the set of modules to which an implementation entity can be mapped.

6. CONCLUSIONS AND FUTURE WORK

Our experience in applying flow equation-style set constraints to refine a design model and the mapping of implementation models to that design model is mostly encouraging. Constraint propagation was effective in extending a handful of manual assignments to almost a complete mapping from source code to model. The completed model provided us valuable information on how to restructure the subject system to provide independent graphical and non-interactive batch user interfaces on a common core of functionality, as we intended. Also, some violations of intended and desirable structure were revealed.

A limitation of the exercise described here is that it involves only a single, small system, about which we had access to extensive (though not complete) design knowledge. We cannot conclude that a similar exercise on a much larger and less familiar system would have been as successful. We are concerned less with the performance and scalability of the the purely mechanical part of the analysis, which is based on well-understood flow analysis algorithms, than with potential growth in the human effort required for larger and less familiar systems. We are currently repeating the exercise with the Apache Tomcat web server [11], a larger system and one with which we are much less familiar.

Our approach depends on provision of an initial model and partial mapping by the user, and it is vulnerable both to errors in the model (e.g., omission of dependence between high-level modules) and errors in the implementation (dependencies that violate the design model). Since small errors in mapping from code to design can have large effects,

practical use of the approach described here will depend critically on dealing with many such small errors together, rather than forcing the human engineer to trouble-shoot each one individually. Problems we have encountered so far were diagnosed with simple analyses. Evaluating and refining techniques for handling inconsistency effectively will be an important facet of future work. Since statistical and heuristic approaches to reverse engineering are more robust with respect to inconsistency, another avenue we will explore is whether and how they might be combined with manipulations based on flow analysis.

The flow equations required not only for the main mapping refinement but also for a variety of other transformations used in the exercise were quite succinct, but they are not easy to read and understand, particularly for someone who is not already well-versed in flow analysis. This may not be a large obstacle to their use in reverse engineering if the same flow equations can be reused to analyze many different systems. Nonetheless, we have become increasingly unhappy with the classic textbook form of flow equations as a means of expression, and we are seeking an alternative that will be less opaque while retaining the expressive power that allowed us to concisely describe these and other transformations.

Acknowledgements
We thank the anonymous reviewers for their many helpful suggestions. An early form of ideas developed in the present work appeared in the M. Sc. thesis of J. Rajamani [17], under the supervision of the second author.

7. REFERENCES

[1] A. Aiken. Introduction to set contraint-based program analysis. *Science of Computer Programming*, 35:79–111, 1999.

[2] O. Ciupke. Autmatic detection of design problems in object-oriented reengineering. In *Technology of Object-Oriented Languages and Systems (TOOLS 30)*, pages 18–32, 1999.

[3] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *Proc. ICSE '92*, pages 138–156. IEEE Computer Society Press, 1992.

[4] M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universitat Berlin, Institut fur Informatik, 2001. Available at `http://bcel.sourceforge.net`.

[5] J. Fabry and T. Mens. Language-independent detection of object-oriented design patterns. *Computer Languages, Systems, and Structures*, 30:21–33, 2004.

[6] H.M. Fahmy and R.C. Holt. Using graph rewriting to specify software architectural transformations. In *15th Conf. on Automated Software Engineering (ASE 2000)*, pages 187–196. IEEE Computer Society Press, 2000.

[7] L. Feijs, R. Krikhaar, and R. Van Ommering. A relational approach to support software architecture analysis. *Science of Computer Programming*, 33:163–212, 1999.

[8] L. Feijs and R. Van Ommering. Relation partition algebra—mathematical aspects of uses and part-of relations. *Software—Practice and Experience*, 28(4):371–400, 1998.

[9] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopolous, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36:564–593, 1997.

[10] J. Fiskio-Lasseter and M. Young. Flow equations as a generic programming tool for manipulation of attributed graphs. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '02)*. ACM Press, November 2002.

[11] Apache Software Foundation. Apache Jakarta Tomcat. Java servlet container available at `http://jakarta.apache.org/tomcat`.

[12] E. R. Gansner, E. Koutsofios, S. C. North, and K. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[13] R. C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *5th Working Conference on Reverse Engineering*, pages 210–219. IEEE Computer Society Press, 1998.

[14] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[15] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[16] D. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(3):128–137, 1979.

[17] J. Rajamani. Constraint propagation as an aid to program visualization. M.Sc. Thesis, Computer Science Dept., Purdue University, 1997.

[18] K. Sartipi and K. Kontogiannis. On modeling software architecture recovery as graph matching. In *Proc. ICSM 03*. IEEE Computer Society Press, 2003.

[19] X. Zhang, M. Young, and J.H.E.F. Lasseter. Electronic appendix to "Refining code-design mapping with flow analysis". Available at `http://www.cs.uoregon.edu/ michal/papers/fse04.html`.