Flow Analysis for Verifying Properties of Concurrent Software Systems*

Matthew B. Dwyer
Department of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506
dwyer@cis.ksu.edu

Lori A. Clarke
Jamieson M. Cobleigh
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
{clarke, jcobleig}@cs.umass.edu

Gleb Naumovich
Department of Computer and Information Science
Polytechnic University
Brooklyn, NY 11201
gleb@poly.edu

Classification

D.2.4 Software/Program Verification, D.1.3 Concurrent Programming

Abstract

This paper describes FLAVERS, a finite-state verification approach that analyzes whether concurrent systems satisfy user-defined, behavioral properties. FLAVERS automatically creates a compact, event-based model of the system that supports efficient data-flow analysis. FLAVERS achieves this efficiency at the cost of precision. Analysts, however, can improve the precision of analysis results by selectively and judiciously incorporating additional semantic information into an analysis.

We report on an empirical study of the performance of the FLAVERS/Ada toolset applied to a collection of multi-tasking Ada systems. This study indicates that sufficient precision for proving system properties can usually be achieved and that the cost for such analysis typically grows as a low-order polynomial in the size of the system.

*This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032, by U.S. Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH01-00-C-R231, by the Advanced Research Projects Agency under Grant MDA972-91-J-1009, the Office of Naval Research under Grant N00014-90-J-1791, the National Science Foundation under Grants CCR-9633388, CCR-9703094, CCR-9708184, and CCR-0093174, and by IBM Faculty Partnership Awards. 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, the Air Force Research Laboratory/IFTD, the U. S. Army, the Office of Naval Research, the U.S. Dept. of Defense, the U.S. Government, the National Science Foundation, or IBM.

1 Introduction

Concurrency is commonly employed to help software systems meet demanding performance or responsiveness requirements. Unfortunately, concurrency often exacerbates the already difficult problem of validating software behavior. In this paper, we describe FLAVERS (FLow Analysis for VERification of Systems), an approach that helps address this problem by verifying whether user-specified properties are valid for all possible executions of a system.

The behavior of a concurrent system may depend on the order in which instructions from different tasks are scheduled relative to one another. Two runs of a system on identical inputs may produce different results if the order of instruction execution differs for these two runs. Consequently, errors may only manifest themselves under a few possible task schedules. To help detect serious problems in concurrent systems, therefore, analysts need tools that can exhaustively consider all possible task schedules and analyze the behavior of the system on those schedules.

To address this concern, researchers have been developing a number of *finite-state verification* approaches. These approaches typically compare a finite model of a system to a property specification. Finite-state verification approaches are not as general as theorem-proving based verification approaches [Hoa69, Dij76] in terms of the kinds of properties that can be proved but, unlike automated theorem provers, they are guaranteed to terminate and require much less mathematical sophistication to use.

FLAVERS is a finite-state verification approach that uses data-flow analysis techniques to verify event-based, behavioral properties of concurrent systems. With FLAVERS, analysts define a set of events that they wish to reason about and specify properties of concurrent systems as patterns of those events. FLAVERS then automatically creates a concise graph model of the system that explicitly represents inter-task communication, synchronization, and event orderings, without enumerating the state space. At the core of FLAVERS is a polynomial-time, conservative flow-analysis algorithm, which determines whether the system satisfies a given property and, if not, provides example traces through the model that violate that property.

As we demonstrate in this paper, it is possible to perform reasonably efficient analyses with FLAVERS. The utility of such analyses should not be judged by efficiency alone, however, but also by the precision of the results. To overcome the traditional imprecision of data-flow analysis, we have developed an approach for incrementally improving precision. With this approach, analysts can examine previous analysis results, decide what information is needed to improve precision, and then introduce constraints that represent this information. Moreover, automated support is provided for creating some common types of constraints.

FLAVERS is applicable to a wide range of concurrency models, as well as sequential systems. In addition to analyzing software implementations, it can be used to analyze high-level architectural descriptions or low-level designs, as long as event-flow information is available. We have conducted a number of case studies [Cha96, NCO96, Dwy97, NACO97] and, in all cases, FLAVERS discovered the event-based errors that were known to exist. In addition, in some cases, FLAVERS discovered previously unknown errors. Industrial users have also reported finding errors using FLAVERS. Many of those errors were simple and were discovered in the process of formalizing properties, however, the more complicated errors were found during verification [BMD96, Cor97].

In this paper we discuss analysis of systems with explicit tasking and rendezvous communication and illustrate our approach using concurrent Ada programs. A toolset, called FLAVERS/Ada, supports most of the constructs in Ada, but as with most static concurrency analysis approaches, does not handle dynamic constructs, such as dynamic memory or task allocation. As is typical for static approaches, the analyst must select a particular system configuration when dealing with dynamic constructs. This and other limitations of FLAVERS are discussed in the conclusion.

Other finite-state verification approaches have primarily focused on hardware descriptions (e.g., [McM93]) or system modeling notations (e.g., [Hol97]). Recently there has been considerable interest in applying finite-state verification to software [VHBP00, BR01, Hol00]. Although promising, these reachability-based approaches are severely limited in the size of the systems that they can evaluate. We believe that FLAVERS makes a number of contributions to flow-analysis and concurrency-analysis research that address some of the limitations of these approaches, including:

- a polynomial-time, data-flow analysis algorithm for proving user-specified properties of a system,
- a concise, event-based model of concurrent systems that is automatically derived from a system representation,
- an incremental process though which analyses of increasing precision can be constructed, and
- a demonstration, through empirical evaluation, that flow analysis can be effectively used to verify concurrent software systems.

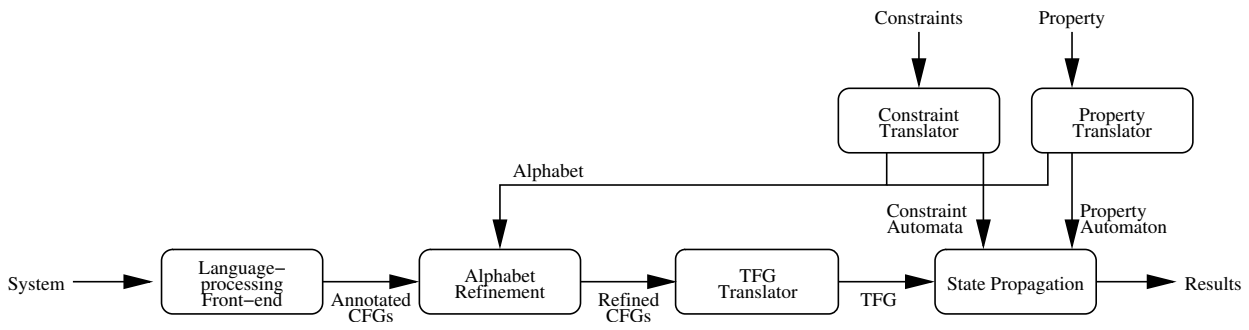

Figure 1: Architecture of FLAVERS

The following section provides a brief, high-level overview of FLAVERS. Section 3 describes the property notations used by FLAVERS, and Section 4 provides a detailed description of the system model. The state-propagation algorithm and its attributes are given in Section 5, followed by a section that describes how constraints are represented and how the flow-analysis algorithm is extended to handle constraints. After having described the FLAVERS analysis approach, Section 7 presents some experimental results gathered from applying the FLAVERS/Ada toolset to the analysis of a collection of concurrent Ada applications. Section 8 discusses related work and Section 9 summarizes the contributions of this approach and discusses limitations and directions for future research.

2 Overview

With finite-state verification approaches, the analyst usually has to experiment with several alternatives before finding an appropriate model of the system that will lead to an effective analysis result. FLAVERS explicitly supports an incremental approach to developing the system model. Using this approach, the analyst provides the software system to be analyzed, the property to be verified, and any initial constraints, such as environmental restrictions, that the analyst knows should be respected. As shown in Figure 1, FLAVERS automatically translates the system, property, and constraints into appropriate internal representations and then employs *state propagation*, a data-flow analysis technique, to determine if the system, as restricted by the constraints, is consistent with the property.

The analysis may return *conclusive* results indicating that the property is consistent with the system. Alternatively, the analysis may return *inconclusive* results; this can happen because of a fault in the system (or in the property) that causes some executable behavior to be inconsistent with the property or because the analysis is too imprecise. FLAVERS generates counterexamples when the analysis returns an inconclusive result. These counterexamples are examined by the analyst and if a fault is apparent, it can be fixed and the analysis can be re-run. In cases where the fault is not obvious or where imprecision in the model is the cause of the inconclusive analysis result, incorporating further constraints can be helpful. This last step of incorporating additional constraints can be performed multiple times, thereby providing an incremental approach for increasing precision.

Specifying Properties: FLAVERS supports the verification of event-based properties. When specifying properties, analysts may wish to reason about the execution of a system from a variety of perspectives. For example, if analysts are interested in determining whether some statement in a system uses a variable whose value is uninitialized, then the events of interest are the definitions and uses of that variable. Or, if interested in determining that whenever a read or write operation on a file is called, the file has been opened and will subsequently be closed, then the events of interest correspond to file operations. To address this variation in focus, analysts define a set of events of interest, where *events* are observable, indivisible system actions. Indivisibility of events is defined with respect to other events. Thus, computation of a complex expression or even a subprogram could be defined as a single event, if it is not possible for intervening events to occur. A set of events used in a FLAVERS analysis is referred to as an *alphabet* and denoted by Σ.

FLAVERS allows analysts to specify properties that describe either desirable or undesirable event sequences, since it is sometimes more convenient to describe a violation of a requirement than to describe the requirement itself.

With FLAVERS, analysts can represent a property as a *quantified regular expression* (QRE) [OO90] or *finite-state automaton* (FSA). From such a description the toolset constructs a *property automaton* that compactly represents the set of event sequences.

Modeling System Executions: To reason about the set of possible system executions, we need a semantically well-founded model of those executions. This model need not represent all the details of the executions, but it must contain sufficient information to support the desired conservative analyses. FLAVERS builds a *trace-flow graph* (TFG) model that compactly represents all sequences of events that correspond to potential system executions for the events of interest. As shown in Figure 1, the events of interest are obtained from both the property and constraints and determine the alphabet for the analysis problem. Each task in the system to be analyzed is first translated into an annotated *control-flow graph* (CFG) and then refined based on this alphabet to form a *refined CFG* (RCFG). After *alphabet refinement*, the RCFGs are combined to form the TFG. The TFG is optimized so that each node in the TFG represents important flow of control information, the execution of some event of interest, or both. The TFG conservatively approximates the executable sequences of events, but may also include some *infeasible sequences*, sequences that occur in the model but do not correspond to any actual executions of the system.

State-propagation Analysis: FLAVERS casts the question of whether system behavior is consistent with a property as a state-propagation, flow-analysis problem [OO90]. We have adapted this approach to apply to concurrent systems and have extended it to incorporate constraints. Conceptually, a sequence of events associated with a path in the TFG would determine a sequence of transitions in the property automaton. The state-propagation algorithm determines all the property states that could be associated with a node in the TFG for all possible paths to that node. This data-flow analysis algorithm avoids enumerating all potential sequences of system events by collapsing them into equivalence classes. The result is an analysis that computes a conservative answer to the analysis question and whose complexity is polynomial in the number of TFG nodes. Conclusive analysis results provide the same assurance as any other formal verification method. Inconclusive analysis results provide information about the source of the faults or about imprecision in the system model. It is important to note that FLAVERS may produce conclusive analysis results even when some infeasible sequences remain in the model. Constraints are only necessary when there are infeasible sequences that cause the property to be violated; elimination of these sequences leads to conclusive results if the actual system does not violate the property.

3 Specifying System Behavior

Finite-state verification compares a model of a system's executable behavior to a specification of its intended behavior. Although it may be possible to specify the intended behavior of a system completely using very expressive formalisms, such as Z [Spi92], the resulting specifications may be as difficult to construct and reason about as the implementation itself. Furthermore, there are no effective means to automate the checking of such specifications against an implementation. In contrast, for finite-state verification, analysts typically write small specifications that capture focused system properties. This eases the burden on the specifier and allows the analysis to be tailored to each property in turn. For a property to be verified by FLAVERS it must first be translated into or expressed directly as a deterministic FSA. This formalism enables FLAVERS to check event-based properties on finite executions[1]. FLAVERS supports both a QRE textual specification and a diagrammatic representation of a quantified FSA.

A QRE consists of an alphabet, a quantifier, and the regular expression to be satisfied. Figure 2 gives the formal syntax for the QRE notation used in FLAVERS. The QRE alphabet contains events that may be used in the regular expression. The meaning of each event symbol is established by the mapping to the appropriate system action, where an occurrence of the symbol represents the execution of the associated action in the system, as described in Section 4.

Quantifiers are included as a convenience. It is often conceptually simpler for an analyst to describe a violation of a property than to describe all of the legal behaviors. The `no` quantifier indicates that no event sequence corresponding to an execution of the system should lie in the language of the specified regular expression; properties using this

[1]FLAVERS has also been extended to support properties on infinite executions using an extended specification notation [NC00]. We do not cover this extension in this paper.

```
qre         :  "for" "events" alphabet "show" expression
                  quantifier "executions"
                  "satisfy" expression

alphabet    :  "{" idlist "}"

quantifier  :  "all" | "no"

idlist      :  id
            |  id "," idlist

id          :  [a-zA-Z][a-zA-Z0-9_:=<>!]*

pos         :  [1-9][0-9]*

:  "."
|  id
|  "~" "[" idlist "]"
|  "[" idlist "]"
|  expression ";" expression
|  expression "|" expression
|  "(" expression ")"
|  expression "^" pos
|  expression "*"
|  expression "+"
|  expression "?"
```

Figure 2: QRE Syntax

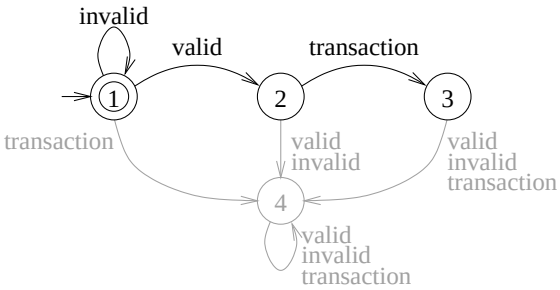

Figure 3: FSA of the ATM property

quantifier specify *undesirable behaviors*. The `all` quantifier indicates that all event sequences corresponding to executions of the system should lie in the language of the specified regular expression; properties using this quantifier specify *desirable behaviors*. Since regular expressions are closed under complement, the expressive power of `all` properties and `no` properties are equivalent.

The `show` clause of a QRE is a statement of the property to be proved and consists of a regular expression defined over the specified alphabet. These regular expressions have the standard meaning given in detail in [OO90]; note that, "." is used to refer to any event in the alphabet, "~" is set complement, ";" is the concatenation operator, "^pos" is used to represent a positive number of repetitions, and "?" means zero or one repetitions.

As an example of a QRE, consider a desirable property of any automated teller machine (ATM) stating that "a valid PIN must be entered before a transaction can take place." Assume the events are `valid` and `invalid`, to indicate when a valid or invalid PIN is entered, respectively, and `transaction`, to indicate when a transaction takes place. The property could then be written as:

```
for events {valid, invalid, transaction}
show all executions satisfy invalid*; (valid; transaction)?
```

which states that any number of invalid PINs may be entered, but a transaction cannot take place until after a valid PIN is entered. Furthermore, if a valid PIN is entered, a single transaction must take place. A variety of other properties specified using QREs are described in Section 7.

FLAVERS also provides support for analysts to directly construct an automaton representation of the property. This form of a property consists of an alphabet and a quantifier, both identical to those used in QREs, and a graphical depiction of the FSA. As is the usual convention, the start state has an incoming arrow with no source and accept states are represented by concentric circles. The automaton form for the above ATM property is illustrated in Figure 3.

The property, whether specified with a QRE or FSA, is translated into a property automaton, an equivalent, canonical form that is a minimal, deterministic, and total FSA. By total we mean that each state must indicate an outgoing

transition for each event in the alphabet. Making an automaton total is done by adding transitions for each missing event from a state to the *trap state*, a non-accepting state with only self-loop outgoing transitions. The trap state and transitions leading in to it are shown in gray in all pictures of property automata. Standard algorithms are used to make the representation minimal and deterministic [ASU85]. Note that there exist expressions for which this translation requires time exponential in the size of the expression but, in our experience, QRE properties tend to be small and so this cost tends to be small compared to the total cost of a FLAVERS analysis. The canonical representation that we use is not required, but simplifies the data-flow algorithm.

Definition 1. A **property automaton** is a deterministic FSA $(\Sigma, S, \delta, A, s, s_{trap})$, where:

Σ is the alphabet of the property,
$S = \{s_0, s_1, \ldots, s_k\}$ is the set of property automaton states that represent equivalence classes of strings over Σ,
$\delta : S \times \Sigma \rightarrow S$ is the total state transition function,
$A \subseteq S$ are the accepting states,
$s \in S$ is the unique start state, and
$s_{trap} \in S$ is the unique trap state.

We use $\mathcal{L}(P)$ to denote the set of all strings accepted by a property automaton P. QREs and their corresponding automata representations are able to succinctly express a wide variety of commonly occurring behavioral properties used in finite-state verification [DAC99].

4 Trace-Flow Graph Model of System Execution

We are interested in reasoning about sequences of events that can occur during execution. Therefore, it is important that our model be a conservative representation of all the possible sequences of these events that could actually occur. On the other hand, a program may manipulate data that can range over an infinite or very large set of values. Thus, in practice, the precise execution behavior of a system cannot be efficiently captured by a finite-state model. Instead, our finite-state model abstracts the system behaviors, while remaining conservative. This model is constructed in a way that preserves all the relevant event sequences, by not modeling all the possible values of variables. Therefore, it is possible that additional event sequences may be represented as well. If this occurs, the analysis will be conservative but may return spurious results. In Section 6, we describe a mechanism by which users can refine the precision of the model, based on the knowledge they have obtained from past analyses. Thus, in considering the trade-off between creating a model that could produce conclusive results but will often be too large to support cost-effective analyses and one that will usually be tractable, but may lead to inconclusive results, we err on the side of the smaller model. In this section we describe the TFG model, demonstrate that this model is easy to construct, and show that it supports conservative state-propagation analysis.

To illustrate the model, we use the small, concurrent Ada program shown in Figure 4. In Ada, a system consists of a set of threads of control, called *tasks*, that may run in parallel. The basic construct for communication and synchronization between tasks is the *rendezvous*, a form of synchronous communication. A task may call on a named *entry* in another task; execution of the calling task is then blocked until the called task accepts the call and the two tasks complete the rendezvous, possibly passing information in both directions. A call that has not yet been accepted is *pending*. A task declaring a particular entry `e` may accept calls from other tasks on that entry by executing an `accept e` statement; if no calls on this entry are pending, the accepting task is blocked until a call on entry `e` is made by another task.

In Figure 4, the system contains two tasks, `customer` and `atm`, that simulate the operation of an ATM. The customer begins by inserting a card, which is represented by the entry call `atm.insert_card`. After a rendezvous with the ATM, the customer enters the PIN, represented by the entry call `atm.enter_pin`. After this rendezvous, the customer is notified by the ATM whether the PIN is valid, invalid, or, in the case where there have been too many invalid PIN entries, that the card is invalid and will be "eaten" by the machine. If the PIN is found to be valid, then the customer can perform a transaction and remove the card from the machine. In this example, `validPIN` represents a

```
task body customer is
begin
  atm.insert_card;
  loop
    atm.enter_PIN;
    select
      accept valid;
      atm.transaction;
      exit;
    or
      accept invalid;
    or
      accept eat_card;
      exit;
    end select;
  end loop;
end customer;
```

```
task body atm is
  count : integer range 0 .. 2;
begin
  count := 0;
  accept insert_card;
  loop
    accept enter_PIN;
    if(validPIN) then
      customer.valid;
      accept transaction;
      exit;
    else
      if(count = 2) then
        customer.eat_card;
        exit;
      else
        customer.invalid;
        count := count + 1;
      end if;
    end if;
  end loop;
end atm;
```

Figure 4: Ada Tasks for the ATM Example

function call that contains no events of interest and thus is treated as an atomic action[2].

4.1 Trace-Flow Graph Construction

A TFG captures all potential event orderings that might occur during execution of the system. The TFG consists of:

- a collection of RCFGs, one for each task in the system
- additional nodes and edges to represent inter-task synchronization and communication actions
- additional edges to represent inter-task event orderings

We consider each TFG component in turn and build up the definition of a TFG in three steps.

4.1.1 Refined Control-Flow Graphs

The RCFG is derived from a typical CFG model [ASU85], which we assume is a conservative approximation of the possible sequences of statements that a task could execute [MR90]. In this section, we define CFGs whose nodes are labeled with events and then introduce a procedure for converting them into RCFGs.

Definition 2. A **CFG** G is a labeled directed graph $(N, n_{initial}, n_{final}, E, label)$, where:
N is the set of CFG nodes,
$n_{initial} \in N$ is the unique initial node of G,
$n_{final} \in N$ is the unique final node of G,
$E \subseteq N \times N$ is the set of directed edges, and
$label : N \rightarrow \Sigma_S \cup \{\tau\}$ is a labeling function that maps a node to its associated event, where Σ_S is the set of all events for a given system and τ is a special, "empty" event.

Tasks can call subprograms, which can in turn call other subprograms. We inline the subprogram's CFG for each

[2]To save space, we do not show the PIN parameter, since it is not needed in the subsequent analyses.

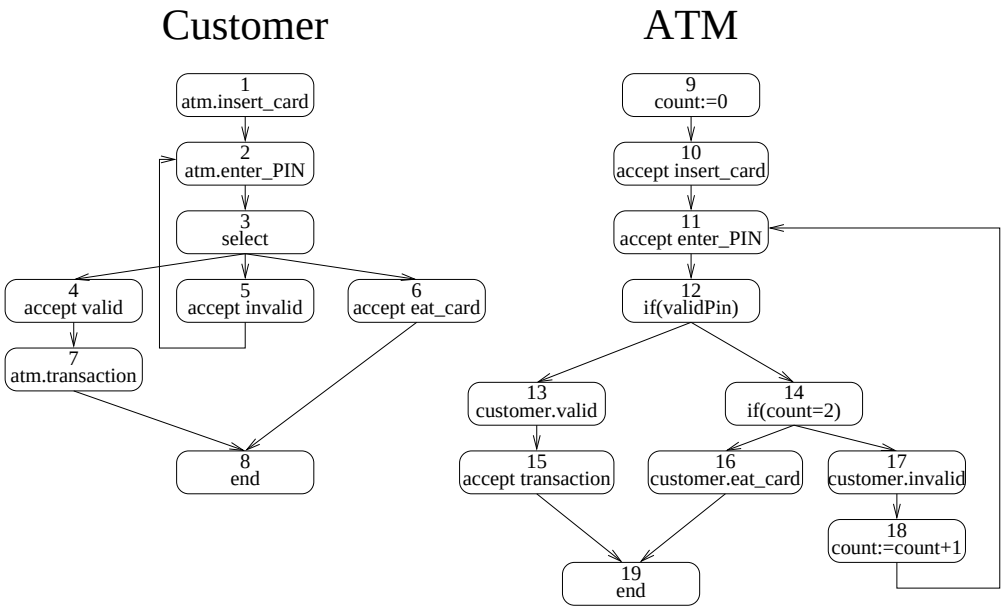

Figure 5: CFGs for the Tasks in Figure 4

called subprogram[3]. Although, in the worst case, this inlining can result in an explosive growth in the size of the graph, we have found that typically this growth is modest when combined with CFG alphabet refinement, described below, since this refinement tends to remove the vast majority of nodes. This significant reduction is not evident in the small example given here, but is illustrated for larger examples in Section 7.

For expediency, an analyst might choose to form a CFG for a set of analyses. Thus, not all events in the system alphabet Σ_S may be needed for a given FLAVERS analysis. Let $\Sigma_I \subseteq \Sigma_S$ represent the actual set of events of interest. This set would contain at least all the events in the property. Also, note that CFGs are affected by the granularity of system events. For example, where the canonical procedure for creating CFGs creates a node corresponding to a basic block [ASU85], our procedure for creating annotated CFGs may create multiple nodes for a basic block, where each node corresponds to a single event in this block. At present, the FLAVERS/Ada tool set recognizes some types of events automatically, such as task communications, but other types of events are indicated by the use of stylized comments placed in the source code.

A CFG conservatively represents each sequence of system events that may be observed on an execution of a task with a path exhibiting precisely the same sequence of events. Figure 5 shows CFGs for the two tasks from Figure 4. For clarity, the nodes of the CFGs are labeled with system events that represent their associated source statement.

Since a rendezvous can only occur in a concurrent system and CFGs are representations of sequential code, the call and accept parts of a rendezvous are each represented by events. These constituent parts are later replaced by the appropriate rendezvous event during the construction of the TFG. Let $\Sigma_C \subseteq \Sigma_S$ contain the set of all inter-task communication events, the labels that correspond to either entry calls or `accept` statements.

We want the model of the system to be as small as possible, yet still be conservative. We therefore transform each CFG into a RCFG, using an *alphabet-refinement* algorithm that eliminates any nodes that are not labeled with an event from Σ_I, the events of interest, Σ_C, the inter-task communication events, or $n_{initial}$ and n_{final}, the initial and final nodes. The alphabet-refinement algorithm is given in Figure 6[4]. This algorithm also relabels with τ the initial and final nodes

[3]The assumption that inlining is performed before construction of RCFGs is made only for simplicity of the presentation. In our implementation, inlining is performed after CFGs are refined. FLAVERS does not currently treat recursive subprograms, although techniques for replacing tail recursion with iteration could be incorporated.

[4]In our implementation of this algorithm we do not remove a node if its removal would increase the number of edges in the RCFG. For example, a node with three predecessors and two successors would not be removed because after this node and its adjacent five edges are removed, six new edges would have to be added to the RCFG.

Input: A CFG $G = (N^G, n^G_{initial}, n^G_{final}, E^G, label^G)$, an alphabet of events of interest Σ_I, and an alphabet of inter-task communication events Σ_C
Output: A RCFG $G' = (N^{G'}, n^{G'}_{initial}, n^{G'}_{final}, E^{G'}, label^{G'})$

begin
  copy G to G'
  let *Retained* $= \{n \in N^{G'} \mid label^{G'}(n) \in \Sigma_I \cup \Sigma_C \vee n = n^{G'}_{initial} \vee n = n^{G'}_{final}\}$
  // remove nodes not labeled with events of interest or inter-task communication events
  forall nodes $n \in (N^{G'} \setminus Retained)$:
    let *Preds* $= \{p \mid (p, n) \in E^{G'} \wedge p \neq n\}$
    let *Succs* $= \{s \mid (n, s) \in E^{G'} \wedge s \neq n\}$
    forall pairs (p, s), where $p \in Preds, s \in Succs$, add edge (p, s) to $E^{G'}$
    remove n and all of its incident edges from G'
  // assign a τ label to nodes not labeled with events of interest or inter-task communication events
  forall nodes $n \in N^{G'}$:
    if $label^{G'}(n) \notin \Sigma_I \cup \Sigma_C$:
      assign $label^{G'}(n) = \tau$
end

Figure 6: CFG Alphabet-refinement Algorithm

of the RCFG, unless they are labeled with an event from $\Sigma_I \cup \Sigma_C$. Thus, after this algorithm terminates, each node in the RCFG is labeled with τ or with an event from $\Sigma_I \cup \Sigma_C$. Note there is a weak bisimulation relationship [Mil89] between a CFG and its RCFG.

To refine the CFGs in Figure 5 to be employed in the verification of the property in Figure 3, the following alphabets are used:

$\Sigma_I = \{$`invalid`, `valid`, `transaction`$\}$

$\Sigma_C = \{$`atm.insert_card`, `atm.enter_PIN`, `accept valid`, `accept invalid`, `accept eat_card`, `atm.transaction`, `accept insert_card`, `accept enter_PIN`, `customer.valid`, `accept transaction`, `customer.eat_card`, `customer.invalid`$\}$

The resulting RCFGs are shown in Figure 7. Note that in this example none of the events in Σ_I occur on any of the nodes of the CFGs or RCFGs since these events correspond to rendezvous events. These events are added to nodes of the TFG when the TFG is created.

To define the safety of RCFG's, we introduce the projection operator $\sigma|_{S'}$ which takes a sequence $\sigma = s_0, s_1, \ldots, s_n$ of elements defined over a domain, $s_i \in S$, and a subset of that domain, $S' \subseteq S$, and returns a sequence $\sigma' = s_{j_0}, s_{j_1}, \ldots, s_{j_m}$ that retains only those elements in σ that are in S', meaning $s_i \in S' \iff (\exists j_k \mid j_k = i)$. Furthermore, the order of elements in the projected sequence is preserved, meaning $j_k < j_{k+1}$.

Theorem 1 (RCFG Conservativeness). *Let G be a CFG and G' the corresponding RCFG obtained using the alphabet-refinement algorithm. Let $R \subseteq N^G$ be the set of nodes of G that were retained in G'. For any sequence of nodes $\pi \in (N^G)^*$, let Map$(\pi) \in (N^{G'})^*$ be the sequence of nodes in G' that corresponds to the nodes in the projection $\pi|_R$. Given this $\pi \in G$ implies Map$(\pi) \in G'$.*

Proof. The proof of this theorem appears in Appendix A.1. □

Once the RCFGs are formed for all the tasks in the system, the first step of the TFG construction is to include the nodes and edges from these RCFGs. We say that these nodes and edges are *local* to their respective RCFG, and thus distinguish the local nodes and edges from the communication nodes and edges that are added by the next step to represent inter-task communication. We refer to the set of edges that connect TFG nodes that correspond to nodes in the same RCFG as E_{local}.

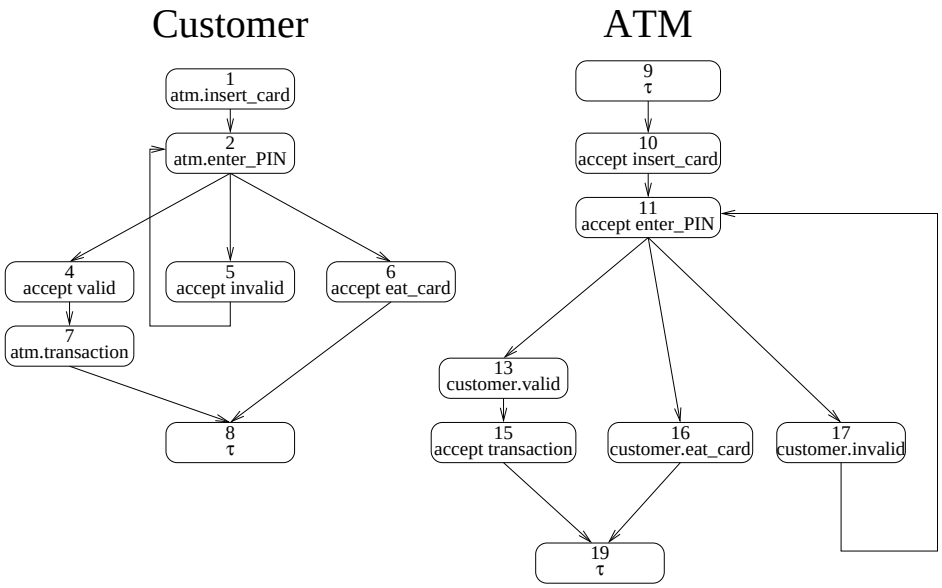

Figure 7: RCFGs, for the CFGs in Figure 5, Refined for the Alphabet of the Property in Figure 3

4.1.2 Task Synchronization

The TFG models all potential synchronization actions by adding a *communication node* between two matching synchronous interactions. A communication node, c, is created for each pair of local nodes n and m from different RCFGs G_i and G_j: $n \in N^{G_i}, m \in N^{G_j}, i \neq j$, where n and m are labeled with events that represent matching communication statements (i.e., one is labeled with an `accept` statement and the other is labeled with an entry call on the same entry). An edge to c is created from each of n and m. In addition, each edge from E_{local} from n or m to a local node s is replaced by an edge from c to s. If the label on n is not in Σ_I, we relabel node n with τ; a similar relabeling is done on node m. If the rendezvous corresponds to an event e in Σ_I, node c is labeled with e, otherwise it is labeled τ. Since a communication node c represents a synchronization between two tasks, we consider node c to be part of both tasks involved in the rendezvous. We use N_{com} to denote the set of communication nodes in the TFG and E_{com} to denote the set of control edges that connect communication nodes to other nodes in the TFG.

In addition to communication nodes, we also create a pair of unique *initial* and *final* nodes for the TFG. The initial node represents the synchronization of tasks at the start of execution. It does not have incoming edges, but has edges to the initial nodes for each of the task RCFGs. The final node represents termination of the system execution. An edge is created from the final node of each task RCFGs to the TFG final node. The TFG final node has no outgoing edges.

Figure 8 shows the TFG for the example in Figure 4. In this TFG representation, communication nodes are shown as diamonds and the initial and final nodes are shown as triangles. For example, RCFG nodes 13 and 4 in Figure 7 represent a call on entry `valid` and an accept of this entry respectively. As shown in Figure 8, node 24 is the communication node that is constructed to represent the task interaction on entry `valid`. Since this rendezvous event is in Σ_I, it is labeled `valid`. Node 21 represents the rendezvous `insert_card`, which is labeled τ since `insert_card` is not in Σ_I. Note that, unlike in the RCFG for task `Customer` in Figure 7, local nodes 1 and 2 are no longer connected by a control edge. This represents the fact that control in task `Customer` can flow from node 1 to node 2 only after completion of the task synchronization represented by the communication node 21.

Different types of task interactions require different patterns of communication nodes. For example, Ada `accept` statements may have *bodies* that consist of executable code. When an entry call is accepted by such an `accept` statement, the calling task blocks while the `accept` body is executed. To represent such interactions, we need two communication nodes, one to represent the start of the entry call and the other to represent the end of that call.

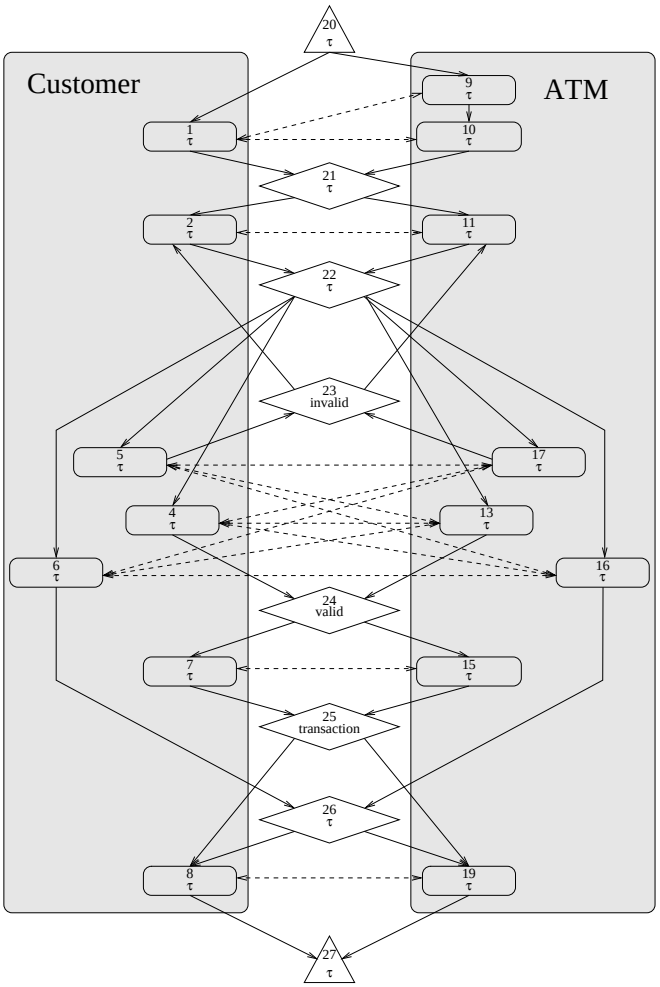

Figure 8: TFG, for the Example in Figure 4, Using the RCFGs in Figure 7

4.1.3 Task Interleavings

The TFG must explicitly represent all possible orderings of events that could occur in concurrently executing tasks. These orderings must be considered during the FLAVERS analysis, and so we add edges (m, n) to the TFG to represent the possibility that execution of code corresponding to node m *may immediately precede* (MIP) execution of code corresponding to node n, where m and n belong to disjoint sets of tasks. Note that we assume an interleaving model of concurrent execution, which means that if two events `a` and `b` in the system may happen in parallel, both sequences `ab` and `ba` must be represented in the model. In Figure 8 MIP edges are shown as dashed arrows (a bidirectional dashed arrow connecting two nodes m and n represents two MIP edges, (m, n) and (n, m)). We denote the set of all MIP edges of the TFG by E_{mip}.

A naïve approach to computing MIP edges for a TFG is to create an edge for all pairs of nodes m and n such that the sets of tasks of m and n are disjoint. Although the resulting set of edges would conservatively capture the MIP relation, many of these MIP edges would connect nodes that in reality could not immediately precede each other. For example, in the TFG in Figure 8, node 11 may not immediately precede node 1, because for task `ATM` to execute node 11, a synchronization with task `Customer` on entry `insert_card` must have happened, which in turn requires that `Customer` completes the execution of node 1. A *spurious* MIP edge, such as $(11, 1)$, leads to imprecision in the analysis, since it introduces paths in the graph that do not correspond to real system executions. In addition, processing

these extra edges increases analysis cost.

Although, in general, the problem of precisely determining which code regions may immediately precede each other is undecidable, it is possible to compute a conservative approximation efficiently. We compute a conservative approximation of the minimal set of necessary MIP edges by using the *may happen in parallel* (MHP) analysis [NA98, NAC99b]. The MHP analysis uses a non-symmetric data-flow analysis algorithm to compute a conservative estimate of pairs of TFG nodes representing regions of code that may happen in parallel with each other. The worst-case complexity of the MHP analysis is cubic in the number of TFG nodes. Based on the MHP information and the task interaction semantics, we compute a set of MIP edges.

4.2 Formalization of the TFG

We construct the TFG from the local nodes of each individual task RCFG with additional communication, initial, and final nodes. The edges of the TFG preserve the local control-flow edges that do not involve task synchronization of the individual RCFGs, but also include edges to represent task synchronization and interleaving explicitly.

Definition 3. Assume that a system is composed of k tasks, where each task T^i has a RCFG $(N^i, n^i_{initial}, n^i_{final}, E^i, label^i)$, where N^i is the set of nodes and $n^i_{initial}$ and n^i_{final} represent the unique first and last node, respectively, $E^i \subseteq N^i \times N^i$ are edges, and $label^i$ relates a node in N^i to an event. A **TFG** is a labeled directed graph $(N, n_{initial}, n_{final}, E, \Sigma_I, label)$ where:

$N = \{n_{initial}, n_{final}\} \cup N_{com} \cup \bigcup_{1 \le i \le k} N^i$, is the set of TFG nodes, where N_{com} is the set of communication nodes,

$n_{initial}$ is the unique initial node,

n_{final} is the unique final node,

$E = E_{com} \cup E_{mip} \cup E'_{local} \cup \bigcup_{1 \le i \le k}\{(n_{initial}, n^i_{initial})\} \cup \bigcup_{1 \le i \le k}\{(n^i_{final}, n_{final})\}$ is the set of TFG edges, where E_{com} is the set of edges between local nodes and communication nodes, E_{mip} is the set of edges representing the interleaving among tasks, and $E'_{local} = \bigcup_{1 \le i \le k}(E^i \setminus R^i)$, where R^i is the set of RCFG edges removed when adding communication nodes,

Σ_I is the set of events of interest defined by the analyst, and

$label : N \rightarrow \Sigma_I \cup \{\tau\}$ maps a TFG node to its associated event where

$$label(n) = \begin{cases} label^i(n) & \text{if } n \in N^i \\ e & \text{if } n \in N_{com},\ e \in \Sigma_I, \text{ and } e \text{ is the event associated with the rendezvous} \\ & \text{represented by } n \\ \tau & \text{otherwise} \end{cases}$$

For the TFG, we define two functions to return the successor and predecessor nodes for a given node n:

$Preds(n) = \{p \mid (p, n) \in E\}$

$Succs(n) = \{s \mid (n, s) \in E\}$

A path in the TFG is assigned meaning via the concatenation of non-τ labels on nodes in the path. Let Π be the set of all paths through the TFG that start with the initial node. We introduce function $Labels : N^* \rightarrow \Sigma_I^*$, which maps a sequence of nodes to a sequence of events in the set Σ_I such that:

$$Labels(n_0, n_1, \ldots, n_k) = label(n_0), label(n_1), \ldots, label(n_k)|_{\Sigma_I}$$

Informally, for a path π, $Labels(\pi)$ is the sequence of events of interest on the nodes in the path. We use $\mathcal{L}(G)$ to denote the set of all event sequences in the TFG:

$$\mathcal{L}(G) = \{Labels(\pi) \mid \pi \in \Pi\}$$

Let $\mathcal{L}(S)$ be the set all event sequences, either partial or full, projected on alphabet Σ_I, of all actual executions of the system S, still assuming an interleaving model of concurrent execution.

Theorem 2 (TFG Conservativeness). *For each actual execution of the system S, the TFG contains a path starting in the initial node that exhibits the same set of events projected on Σ_I as this execution projected on Σ_I. Thus, $\mathcal{L}(S) \subseteq \mathcal{L}(G)$.*

Proof. A formal proof of this theorem appears in Appendix A.2. □

Theorem 3 (Size of TFG). *The number of nodes in a TFG, $|N|$, is $\mathcal{O}(S^2)$ where S is the sum of the number of nodes in the task RCFGs. The number of edges in a TFG is $\mathcal{O}(S^4)$.*

Proof. Let C be the number of entry call nodes and A the number of accept statement nodes in the task RCFGs. The number of nodes in the TFG is $\mathcal{O}(S + A * C + 2)$. This is the sum of the number of RCFG nodes, communication nodes, and 2 for the initial and final nodes. In the worst case, if $C \approx A \approx S/2$, the bound on $|N|$ becomes $\mathcal{O}(S^2)$.

A TFG has at most two directed edges between any pair of nodes. Therefore, the number of TFG edges is in the worst case $\mathcal{O}(|N|^2) = \mathcal{O}(S^4)$. □

The product of the number of entry calls and accept statements is more accurately computed on a per entry basis and then summed over the set of entries; this value can be significantly less than the product of all entry call and accept statements. In addition, communication statements are typically a small percentage of the statements in a given system. Consequently, we have found that the worst-case quadratic blow-up in the number of TFG nodes compared to the total number of RCFG nodes is not common in practice. In the experimental study reported here and in an experimental study of 159 mostly small Ada programs [NA98], it was found that the nodes labeled with events from Σ_I tend to be sparse in the graph and that the number of TFG nodes tends to be linear in the number of RCFG nodes and, thus, linear in the number of statements in the program.

Theorem 4 (Complexity of the TFG Construction Algorithm). *The time required to construct a TFG from a collection of CFGs is $\mathcal{O}(S^6)$, where S is the sum of the number of nodes in the task RCFGs.*

Proof. The TFG construction consists of several steps. First, RCFGs are constructed, which takes $\mathcal{O}(S^2)$ time. Next, the initial, the final, and the communication nodes and then the MIP edges are added. Since there are at most $\mathcal{O}(S^2)$ communication nodes, adding communication nodes takes $\mathcal{O}(S^2)$ time. MHP analysis is $\mathcal{O}(N^3)$, where N is the number of TFG nodes and therefore is $\mathcal{O}(S^6)$. The procedure for adding MIP edges is linear in the number of MHP pairs, which is $\mathcal{O}(N^2) = \mathcal{O}(S^4)$. Thus, the complexity of constructing the TFG is dominated by the complexity of the MHP analysis, which is $\mathcal{O}(S^6)$. □

In our experiments, the time to construct the TFG was at most cubic in the size of the system, as described in Section 7.

4.3 Reducing the size of the TFG

Clearly, the size of the TFG has a major impact on the cost of the analysis. CFG refinement tends to reduce the size of the graph significantly. In addition, there are several other optimizations that can be employed.

One such optimization is partial-order reduction, which may remove some of the MIP edges from the graph, using an algorithm that takes into account partial orders of events from different tasks [NCC99]. This algorithm is conservative in the sense that a MIP edge is only removed if it can be shown that for each possible event sequence in the original TFG, there is an event sequence in the reduced TFG that has an identical effect on the property. Note that although the initial computation of MIP edges based on the MHP analysis does not depend on the alphabet, this partial-order reduction does. In our evaluation of this optimization, the partial-order reduction on average removed 25% of the MIP edges and improved the analysis time by about 20% on the examples where this optimization was applicable [NCC99].

We can use the MHP analysis to further reduce the size of the TFG since in some cases it can determine that a communication node represents a synchronization that can never happen. Let nodes m and n correspond to an entry call and an `accept` on the same rendezvous. As described in Section 4.1.2, the TFG construction algorithm creates a communication node c with m and n as predecessors. A necessary condition for an inter-task communication is the possibility that the entry call and the accept statement execute simultaneously. Therefore, if the MHP analysis determines that m and n may not happen in parallel, it means that the rendezvous represented by c can never occur.

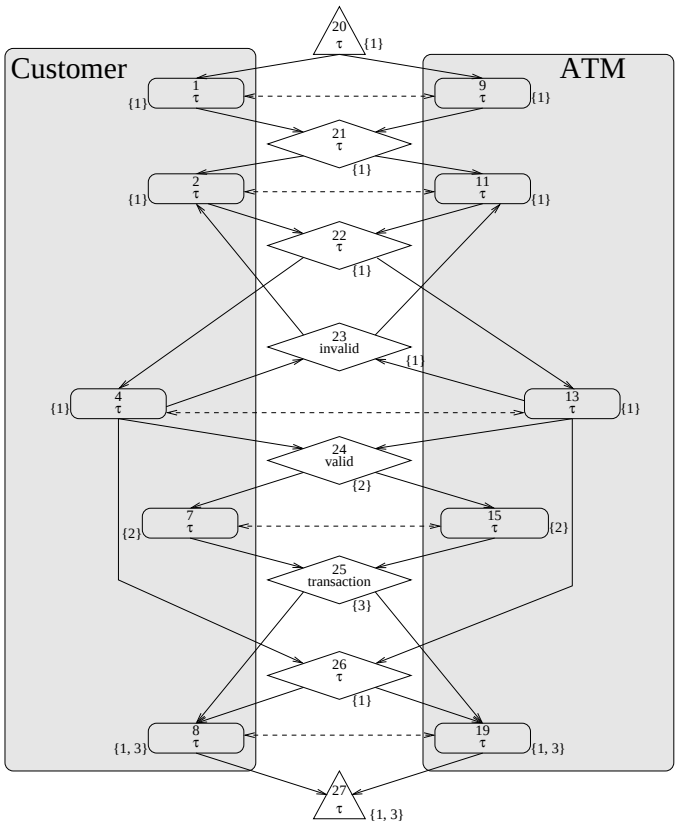

Figure 9: Optimized TFG, Annotated with States Obtained from State Propagation of the Property in Figure 3

We can remove node c from the TFG. After removing all non-executable communication nodes and their incident edges, we can do a pass through the TFG removing all the nodes that cannot be reached using edges in $E \setminus E_{mip}$.

We perform additional conservative structural optimizations during TFG construction, but do not included them in the formal description since they complicate the discussion about conservativeness and complexity bounds. The TFG construction algorithm, as presented, may add unnecessary τ nodes to the TFG. For example, in the TFG in Figure 8, node 9 was retained because it was the initial node in the ATM task and node 10 was retained because it corresponded to an `accept` on the `insert_card` rendezvous. Since nodes 9 and 10 are τ nodes, they have no effect on the state of the property and one of these nodes can be removed from the TFG without changing the set of non-τ event sequences in the TFG.

In addition, notice that node 22 has three successors in task `Customer` that are τ nodes, nodes 4, 5, and 6. We can replace these three nodes by a single node with an out-edge to every node that has an out-edge from node 4, 5, or 6. A similar optimization could be made to collapse nodes 13, 16, and 17 into a single node. Neither of these optimizations affect the set of non-τ event sequences associated with a TFG.

Figure 9 shows the resulting TFG after applying these last two optimizations to the TFG shown in Figure 8. There are some additional annotations on the nodes that can be ignored for now. In this TFG, node 10 has been removed; nodes 4, 5, and 6 have been replaced with a single node, numbered 4; and nodes 13, 16, and 17 have been replaced with a single node, numbered 13.

5 State-propagation Analysis

FLAVERS uses state propagation to compare the executable behavior of a system with event sequences defined in a property. Howden [How86] and later Olender and Osterweil [OO92] developed state-propagation algorithms for checking properties of sequential systems; our work builds on these results. Here, we extend the state-propagation analysis for sequential systems [OO92] to concurrent systems modeled as a TFG.

Let $P = (\Sigma^P, S^P, \delta^P, A^P, s^P, s^P_{trap})$ be a property automaton and let $G = (N^G, n^G_{initial}, n^G_{final}, E^G, \Sigma^G, label^G)$ be a TFG. Recall that $\mathcal{L}(P)$ is the language of the property, the set of all event sequences accepted by P. Let $\mathcal{L}(G^{\text{term}}) \subseteq \mathcal{L}(G)$ be the set of all event sequences found on paths that start at the initial node of G and end at the final node of G. We check consistency between the property and system by checking either language containment, $\mathcal{L}(G^{\text{term}}) \subseteq \mathcal{L}(P)$, if the property specifies desirable behaviors, or empty language intersection, $\mathcal{L}(G^{\text{term}}) \cap \mathcal{L}(P) = \emptyset$, if the property specifies undesirable behaviors of the system.

Intuitively, the objective of state-propagation analysis is to check each path π through the TFG against the property automaton P. If P represents desirable behaviors of the system, then for each path π, the corresponding event sequence $Labels(\pi)|_{\Sigma^P}$ has to be accepting for P to hold for the TFG. If P represents undesirable behaviors of the system, then for each TFG path π, the corresponding event sequence $Labels(\pi)|_{\Sigma^P}$ must not be accepting for P to hold. Since the number of paths in a TFG may be infinite, we use data-flow analysis to collapse paths into equivalence sets with respect to a property P.

We define state-propagation analysis as an instance of a data-flow framework comprised of a semi-lattice of values and a set of functions defined over those values [Hec77]. The values in the analysis are organized as a join semi-lattice L^P over the power-set 2^{S^P} of P's states. The bottom value $\perp$ for this lattice is $\emptyset$ and the top value $\top$ is S^P. The join operation is set union, $\cup$, and the ordering relation is subset, $\subseteq$.

For each node n in the TFG we define a *propagation function* $F^n : 2^{S^P} \rightarrow 2^{S^P}$ that captures the effect that this node has on the property automaton:

$$\forall n \in N^G, \forall S \subseteq S^P, F^n(S) = \left\{ s' \in S^P \mid s \in S \wedge s' = f^P(n, s) \right\}$$

where

$$f^P(n, s) = \begin{cases} \delta^P(s, label^G(n)) & \text{if } label^G(n) \in \Sigma^P \\ s & \text{if } label^G(n) \notin \Sigma^P \end{cases}$$

Given a flow graph that represents all executable sequences of system events, we can instantiate a data-flow framework for it and be sure that its solution contains a conservative set of property automaton states at each flow-graph node [MR90]. Informally, this means that the set of states associated with a particular node in the TFG includes all the property states that could be reached along an event sequence corresponding to a path through the TFG from the initial node up to that node. Since the TFG is a conservative representation of all sequences of the events of interest that could occur during execution, the states associated with each node are also a conservative representation of all the property states that could be associated with the system location that is represented by this node.

The solution at the final TFG node can be compared to the set of accepting states of P. If P specifies desirable behaviors, it holds if the set of states computed for the final node of the TFG contains only accepting states. If P specifies undesirable behaviors, it holds if the set of states computed for the final node of the TFG contains only non-accepting states.

Theorem 5 (State-propagation Conservativeness). *If the state-propagation analysis over a TFG determines that a property holds, then this property holds for the actual system.*

Proof. This proof can be found in Appendix A.3. □

Formulation of our analysis as a monotone data-flow framework provides an algorithm for its solution. Unfortunately, a straightforward implementation of the data-flow framework as an instance of Hecht's iterative worklist algorithm [Hec77] is not very efficient, as it has the worst-case complexity of $\mathcal{O}(|N^G||E^G||S^P|^2)$.

We can improve the worst-case complexity of the state-propagation algorithm significantly by optimizing the propagation of states among the TFG nodes. In particular, we would like to ensure that for any two nodes n and p, where

$p \in$ *Preds*(n), if a state s is associated with node p, then n's transition function F^n will be applied to s only once. We achieve this by associating two sets of states with each TFG node, set *States*(n), which contains all states associated with node n, and set *IN*(n), which contains all states that have been propagated to n from n's predecessors. On each iteration of the algorithm, we update both sets for a single node n. For each predecessor p of n, we add to *IN*(n) only those states from *States*(p) that have not been placed in *States*(n) on a prior iteration and apply the transition function of n to only those states, placing the resulting states in *States*(n). Figure 10 shows the FLAVERS state-propagation algorithm incorporating this optimization.

Theorem 6 (State-propagation Complexity). *Given a TFG, with nodes N^G, and a property automaton, with states S^P, the state-propagation algorithm terminates in $\mathcal{O}(|S^P||N^G|^2)$ time.*

Proof. This proof is given in Appendix A.4 □

5.1 Examples

To illustrate the state-propagation algorithm, we use the TFG in Figure 8, optimized as described in Section 4.3, and the property automaton in Figure 3. The result of state propagation is shown in Figure 9.

State propagation begins by associating the start state of the property automaton, state 1, with the initial node of the TFG, node 20. The *Flow* sets of the two out edges of the initial node, $(20, 1)$ and $(20, 9)$, are initialized to the set containing the start state of the property automaton, $\{1\}$. The successors of node 20, nodes 1 and 9, are then added to the worklist. Suppose node 1 is removed from the worklist first. State propagation next computes the set of property automaton states that need to be propagated over node 1. This is the union of the *Flow* sets of the in edges to node 1 minus the property automaton states that have already reached node 1. This is $(\mathit{Flow}[(20, 1)] \cup \mathit{Flow}[(9, 1)]) \setminus \mathit{IN}[1] = (\{1\} \cup \emptyset) \setminus \emptyset = \{1\}$. The *IN* set of node 1 is next updated to reflect this, so $\mathit{IN}[1] = \{1\}$. Since the property automaton states associated with the in edges of node 1 have been processed, the *Flow* sets of these edges are reset to the empty set. Since node 1 is a τ node, propagating state 1 of the property automaton across it does not change the state of the property automaton. Since state 1 is not in *States*$[1]$, state 1 needs to be propagated to the successors of node 1, so $\mathit{Flow}[(1, 9)] = \{1\}$, $\mathit{Flow}[(1, 21)] = \{1\}$, and nodes 9 and 21 are added to the worklist. Since state 1 has been processed on node 1, it is added to *States*$[1]$, so *States*$[1] = \{1\}$.

Now, nodes 9 and 21 are on the worklist. Suppose node 9 is removed from the worklist next. The processing of node 9 is similar to the processing of node 1, and results in state 1 being associated with node 9, $\mathit{Flow}[(9, 1)] = \{1\}$, $\mathit{Flow}[(9, 21)] = \{1\}$, and nodes 1 and 21 being added to the worklist. When node 1 gets removed from the worklist, the set of states that needs to be propagated over node 1 will be computed again, which is $(\mathit{Flow}[(20, 1)] \cup \mathit{Flow}[(9, 1)]) \setminus \mathit{IN}[1] = (\emptyset \cup \{1\}) \setminus \{1\} = \emptyset$, meaning no new states need to be propagated over node 1. Thus, the use of *Flow* sets has prevented state propagation from processing states that have already been processed on a node.

At this point, only node 21 is on the worklist. State propagation will process nodes 21, 2, 11, 22, 4, and 13, in a manner similar to the processing of nodes 1 and 9, and will associate state 1 with each of these nodes. After processing node 4 (and 13), state 24 will be on the worklist, with $\mathit{Flow}[(4, 24)] = \{1\}$ and $\mathit{Flow}[(13, 24)] = \{1\}$. This results in state 1 being propagated over node 24. Since node 24 is labeled with `valid`, which causes the property automaton to transition from state 1 to state 2, state 2 will be added to $\mathit{Flow}[(24, 7)]$, $\mathit{Flow}[(24, 15)]$, and *States*$[24]$. State propagation will continue until a fixed point is reached, at which time states 1 and 3 of the property automaton will be in *States*$[27]$. Since node 27 is the final node of the TFG, the property being checked is an `all` property, and states 1 and 3 are both accepting states of the property automaton, FLAVERS would conclude that the property holds on all executions.

Figure 11 shows an `all` property automaton stating that "the ATM will eat a card only if an invalid PIN is entered two times in a row." To check this property, FLAVERS produces a new TFG that contains events relevant to this property. This TFG, including the annotations that would result from state propagation, is shown in Figure 12.

In this case, state propagation reports an inconclusive result. This is because non-accepting state 4 of the property automaton in Figure 11 has been propagated to the final node, node 27. FLAVERS would then present the user with a counterexample path through this TFG, for example 20, 9, 21, 11, 22, 13, 26, 19, 27. This path corresponds to the event sequence `eat_card`. This path is spurious, because along this path the variable `count` is first set to 0, and then

Input:
A TFG $G = (N^G, n^G_{initial}, n^G_{final}, E^G, \Sigma^G, label^G)$ and a property automaton $P = (\Sigma^P, S^P, \delta^P, A^P, s^P, s^P_{trap})$

Auxiliary data structures:

States	is an array of length $\lvert N^G \rvert$ that holds sets of states for each node
W	is a worklist represented as a set of nodes, where at all times $W \subseteq N^G$
IN	is an array of length $\lvert N^G \rvert$ that holds sets of incoming states for each node
Flow	is an array of length $\lvert E^G \rvert$ that holds sets of states for each edge.

Initialization:

$$\forall n \in N^G, \mathit{States}[n] = \begin{cases} \{s^P\} & \text{if } n = n^G_{initial} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{IN}[n] = \emptyset$$

$$\forall (n,r) \in E^G, \mathit{Flow}[(n,r)] = \begin{cases} \{s^P\} & \text{if } n = n^G_{initial} \\ \emptyset & \text{otherwise} \end{cases}$$

$$W = \mathit{Succs}(n^G_{initial})$$

Main Loop:
We evaluate the following statements repeatedly until $W = \emptyset$:

Remove a node n from W	/* Remove a node from the worklist */
$V = \bigcup_{p \in \mathit{Preds}(n)} \mathit{Flow}[(p,n)]$	/* Compute the set of all new property states that need to be propagated through n. The union of the *Flow* sets of n's predecessors are the property states that are new to n's predecessor nodes */
$V = V \setminus \mathit{IN}[n]$	/* Remove the states that have already been propagated through n */
$\mathit{IN}[n] = \mathit{IN}[n] \cup V$	/* Update the *IN* set of n so it contains the new property states */
$\forall p \in \mathit{Preds}(n), \mathit{Flow}[(p,n)] = \emptyset$	/* Clear the flow sets of n's in edges */
$U = F^n(V) \setminus \mathit{States}[n]$	/* Propagate the states in U through n, removing property states that have already been encountered */
if $U \neq \emptyset$ then	/* If there are any new property states, i.e. U is not empty */
$\forall r \in \mathit{Succs}(n), \mathit{Flow}[(n,r)] = \mathit{Flow}[(n,r)] \cup U$	/* Add the new property states to the *Flow* sets of n's out edges */
add nodes in $\mathit{Succs}(n)$ to W	/* Add n's successor nodes to the worklist */
$\mathit{States}[n] = \mathit{States}[n] \cup U$	/* Update the set of states encountered on n to contain the property states in U */

Determining the analysis results:

if P represents desirable behaviors then
  if $\mathit{States}[n^G_{final}] \subseteq A^P$ then
    report conclusive result: the property holds
  else
    report inconclusive result: the property may not hold

if P represents undesirable behaviors then
  if $\mathit{States}[n^G_{final}] \cap A^P = \emptyset$ then
    report conclusive result: the property holds
  else
    report inconclusive result: the property may not hold

Figure 10: State-propagation Algorithm

the true branch of the statement `if(count=2)` is taken. This occurs because the value of the variable `count` has not been modeled in the analysis. In the next section, we describe how to model variables when they are deemed to be important to the analysis.

Although not shown here, FLAVERS provides support for selecting short paths that violate the property and for visualizing paths. Path visualization is particularly important when dealing with medium size or large programs or concurrent systems of just about any size.

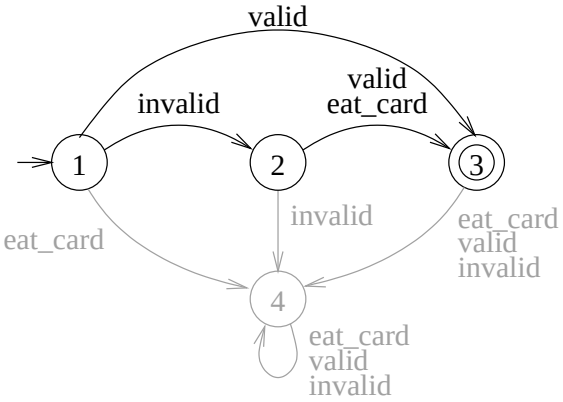

Figure 11: Property Stating that a Card Is Eaten Only After an Invalid PIN Has Been Entered Two Times in a Row

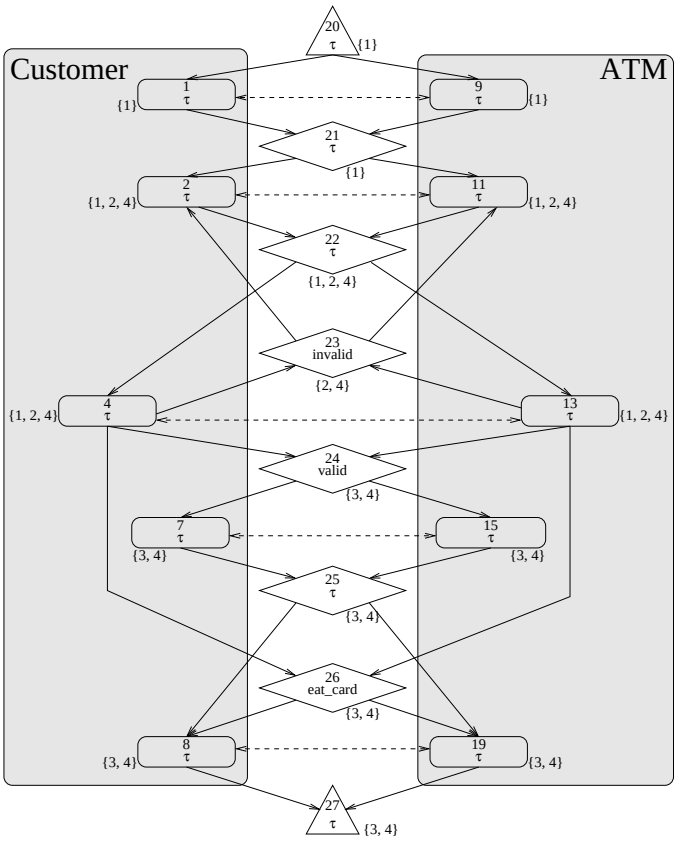

Figure 12: TFG, for the Example in Figure 4, Refined for Alphabet {`valid`, `invalid`, `eat_card`} and Annotated with States Obtained from State Propagation of the Property in Figure 11

6 Constraints

FLAVERS, like all practical flow-analysis approaches, is based on abstractions of system control and data information that are encoded into the analysis. The term sensitivity is often used to describe whether a kind of information

is (partially) preserved by the abstractions used in a flow analysis. Flow-sensitive and context-sensitive analyses honor, respectively, the sequencing of statements and proper nesting of procedure invocations during system execution. Synchronization-sensitive analyses honor the semantics of synchronization statements during execution of a concurrent system [Ram00]. Path-sensitive analyses improve on flow-sensitive analyses by incorporating some path feasibility information. Typically, path-sensitivity is achieved by defining a value-sensitive analysis that captures partial information about program variables (e.g., [HR81]). Increased sensitivity in an analysis leads to increased precision in analysis results. Unfortunately, enforcing even a few forms of sensitivity, such as flow, context, and synchronization sensitivity, throughout the program in a single analysis makes most interesting analysis problems undecidable [Ram00]. The key to efficient and effective flow analysis is in targeting the sensitivity of the analysis to parts of the program that are related to the analysis question.

Figure 12 illustrates the need for increased sensitivity in a FLAVERS analysis. For the example program, a property that is concerned with the number of times an invalid PIN has been entered would need to monitor the possible values of variable `count`. The CFG, and consequently the RCFG and TFG, would normally not include such information. As a result, if the state-propagation analysis determines that a property is violated, it is possible that the violation occurs only on infeasible paths. Since excessive spurious violations hinder usability of a tool such as FLAVERS, it is important to develop techniques for improving the analysis precision without sacrificing too much efficiency.

In this section, we describe a flexible mechanism for enforcing different forms of sensitivity in a FLAVERS analysis. FLAVERS allows information that is not explicitly represented in the TFG to be added to an analysis in the form of *constraints*. Constraints are selectively introduced in situations where it is determined that the analysis results are too imprecise without this additional information. If a constraint is determined to be violated along some TFG path, then the effect of that path on the analysis is discarded. Constraints can encode information, such as statement sequencing, variable values, and branch decisions, that can be used to remove infeasible paths and, thus, to increase the sensitivity of a FLAVERS analysis.

In FLAVERS, constraints are represented as FSAs whose transitions are based on events associated with TFG nodes. Each constraint has a single *violation* state that represents the fact that an event sequence leading to that state is inconsistent with the constraint. Intuitively, if on a path through the TFG a constraint automaton enters a violation state, this path is recognized as spurious and removed from consideration. Similar to the state-propagation analysis defined for property automata in Section 5, we define a combined state-propagation analysis to form a qualified flow analysis [HR81].

Definition 4. A **constraint automaton** is a deterministic FSA $(\Sigma, S, \delta, A, s, v)$, where:

Σ is the alphabet of the constraint,
S is the set of states that represent equivalence classes of strings over Σ,
$\delta : S \times \Sigma \rightarrow S$ is the total state transition function,
$A \subseteq S$ are the accepting states,
$s \in S$ is the unique start state, and
$v \in S$ is the unique *violation* state.

There are several different types of constraints, including variable, task, environmental, and interface. Since constraints can be added for a subset of program features (e.g. a subset of program variables), FLAVERS provides the ability to define partially sensitive flow analyses that are optimized to retain only the information that is necessary to achieve a desired level of precision. Possible FLAVERS analyses range from an unconstrained analysis, which is flow insensitive due to the fact that MIP edges introduce inter-task paths that violate control flow within a task, to a fully task and variable constrained analysis, which is flow, synchronization, scalar-value, and path sensitive.

Although variable and task automata are the only kinds of constraints for which FLAVERS currently provides automated support, the concept is very general; a wide range of finite-state abstractions could be encoded as constraints and incorporated into FLAVERS using state-propagation analysis. We discuss this further in Section 8. User interaction profiles [BMD96], models of the networking environment [NCO96], and models of the software environment [Dwy97] have each been incorporated into FLAVERS analyses using constraints.

In the remainder of this section we describe variable automaton constraints and task automaton constraints in more detail. We then discuss our technique for combining multiple constraints into a single analysis.

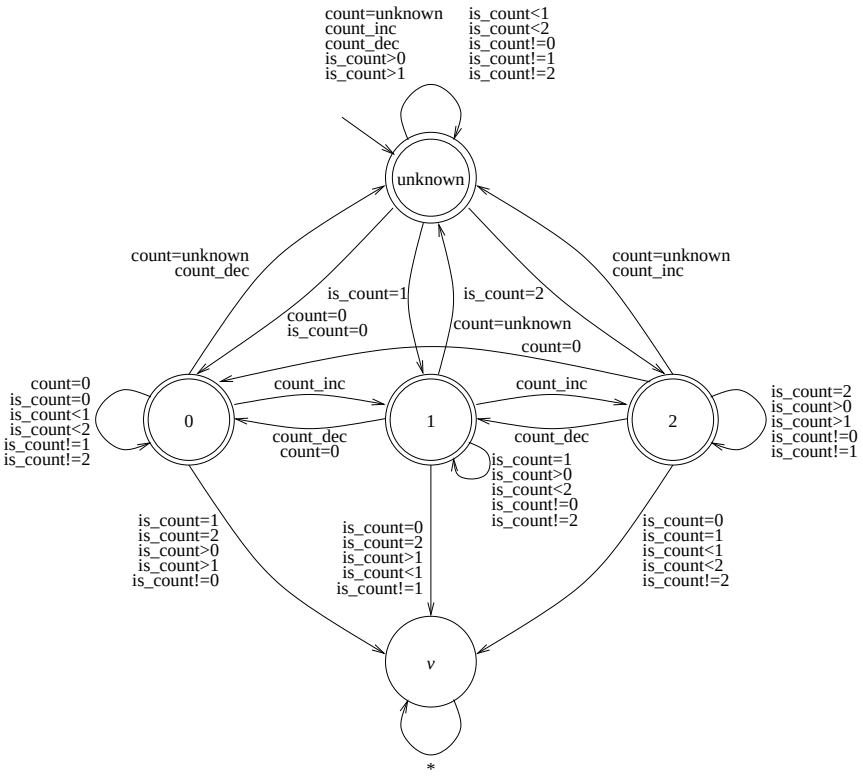

Figure 13: VA for Variable `count`

6.1 Variable Automaton Constraints

Flow graphs do not typically model system variables. For many systems, however, accurate analysis depends on modeling some data values. As the example in Figure 12 illustrates, often we are interested in modeling variables that are used in conditional statements or in guards that control task communication statements. Many of these variables are defined over small finite domains and modified in a disciplined way. Examples include Boolean variables to which only constant values are assigned and bounded integer counter variables to which only increment and decrement operations are applied. We have developed automated techniques for encoding what is known about the value of a variable as *variable automaton* (VA) constraints.

VA transitions represent modifications to the value of the variable and the results of conditional tests of the values of those variables. The violation state, v, represents that a path in the TFG violates the semantics of a branch decision or variable modification with respect to the current variable value.

To reduce imprecision when checking the property in Figure 12, we build a VA for the counter variable `count`, as shown in Figure 13. In this VA, selected operations on a variable are interpreted precisely, for example, assignment of the constant zero and increment and decrement operations for values between zero and two; all other operations are safely abstracted to the situation where the variable's value is non-zero or unknown. Assignment of zero to the variable is modeled as `count=0`; we include `count=unknown` to represent situations where the assigned value is unknown. Conditional results are represented by the possible outcomes (`is_count=0`, `is_count!=0`, and `is_count>1`). There are transitions to the violation state when the result of a conditional is inconsistent with the value of the variable described by the VA state. For example, the test `is_count=0` is false if the value of count is known to be 1, so there is a transition from 1 to the violation state on this event. The self-transition out of the violation state is labeled with a * to indicate that this transition is labeled with every possible event in the alphabet. The structure for other counter VAs is similar, although each will vary in size depending on the range of permissible values.

To use this VA, events modeling the behavior of `count` need to be added to the TFG, as shown in Figure 14. In this

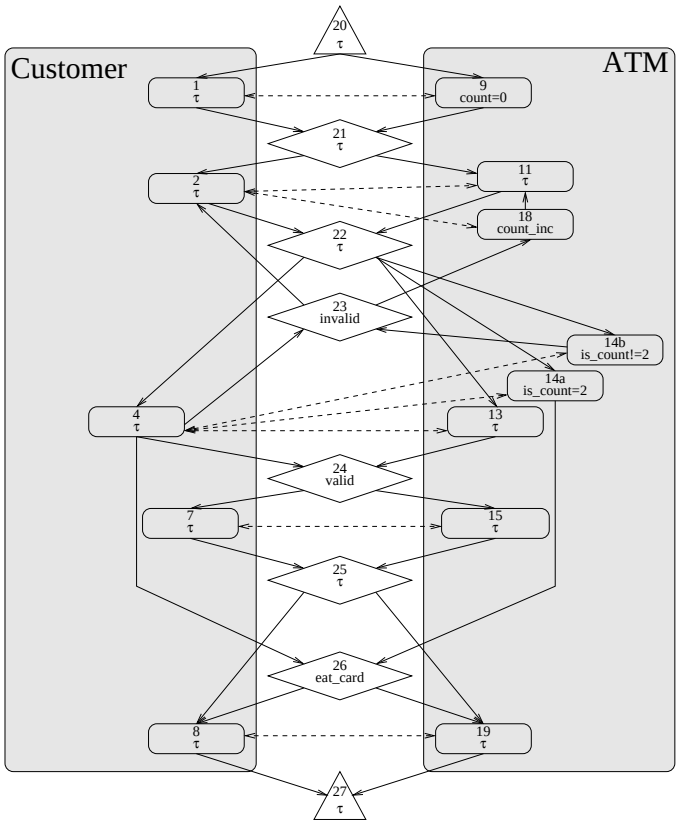

Figure 14: TFG, for the Example in Figure 4, Refined for Alphabet {`invalid`, `valid`, `eat_card`, `count=0`, `count_inc`, `is_count=2`, `is_count!=2`}

TFG, node 9 represents the initialization of variable `count` to 0, node 18 represents the increment of this variable, and nodes 14a and 14b represent the comparison of this variable to 2. Nodes 14a and 14b ensure that the true branch (false branch) is taken only when the condition is true (false).

Consider the path 20, 9, 21, 11, 22, 14a, 26, 19, 27 in the TFG in Figure 14 that corresponds to the infeasible path in the TFG in Figure 12 that we considered earlier. This path corresponds to the sequence of events `count=0`, `is_count=2`, `eat_card`. The subsequence `count=0`, `is_count=2` that contains events from the alphabet of the VA for `count` causes this VA to enter its violation state, which means that this path is infeasible, and, as explained in Section 6.3 below, this path will be discarded by FLAVERS during state propagation using this VA.

To enforce the conditions encoded in a VA during state propagation, the TFG alphabet must include the events in the VA alphabet and the TFG must be extended to identify system statements that cause variable state transitions. In many cases this could be done automatically, although the FLAVERS/Ada toolset currently requires that the events associated with a VA be added through the use of stylized comments in the source code, as mentioned previously.

Counter, enumerated, and Boolean VAs are constructed in $\mathcal{O}(k^2)$ steps, where k is the number of different possible values the variable can assume. The FLAVERS toolset allows users to select a VA definition, then automatically specializes that definition for a specific variable by instantiating a generic VA template and replacing the variable names in the template with the system variable name.

6.2 Task Automaton Constraints

Although MIP edges in the TFG are used to model the possible interleaving of events from different tasks succinctly, they have the negative side effect of introducing paths that may violate event orderings that are encoded as control-flow

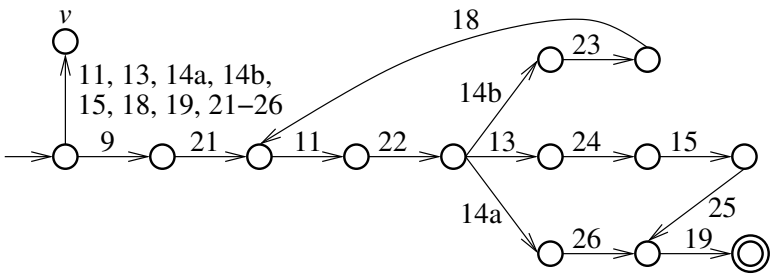

Figure 15: TA for Task ATM, built for the TFG in Figure 14. (For the violation state, all incoming transitions, except for those coming from the start state, are omitted.)

edges in the TFG. For example, in Figure 14 the analysis may propagate data-flow information through the TFG nodes in the following order: 20, 9, 21, 2, 18, 2, 18, 11, 22, 14a, 26, 19, 27. This path violates the control flow in task ATM in several ways, for example node 18 appears on this path before node 23 does, although the control flow through task ATM indicates that node 23 must precede node 18. Therefore, this path is infeasible and its removal would improve the analysis precision, since the property in Figure 11 is violated on this path; note that the VA for variable count is not violated on this path.

Infeasible paths that violate the control flow of a task are removed by encoding the control flow of that task as an FSA, called a *task automaton* (TA). During state propagation, this TA constrains the analysis to consider only those TFG paths that include events from that task in an order that is consistent with its intra-task control flow.

To construct the TA for a task, we build the node-edge dual of the task's sub-flow-graph. Although we could build this from the RCFG directly, we use the TFG so any refinements that may have been applied to the TFG will be reflected in the task's sub-flow-graph as well.

We use function *Tasks* to return a set of tasks to which a TFG node belongs. For any local node, *Tasks* returns a set containing a single task and for any communication node, *Tasks* returns a set containing two tasks whose rendezvous is represented by this communication node. A TA is constructed for task T_i from the sub-flow-graph of the TFG with the set of nodes $\{n|n \in N \wedge T_i \in \mathit{Tasks}(n)\}$ and the set of edges in this sub-flow-graph that are incident upon these nodes. The initial and final nodes of the sub-flow-graph determine the start and accept states of the TA, respectively.

Figure 15 shows the TA for task ATM from the TFG in Figure 14. Each transition in this TA corresponds to executing code represented by a TFG node and each state represents the task state before or after such execution. For clarity, we only show those transitions to the violation state that leave the start state of this automaton. For any node from task ATM, there is a transition based on this node from every state in the TA. All of the transitions that are not shown lead to the violation state.

Consider again the TFG path 20, 9, 21, 2, 18, 2, 18, 11, 22, 14a, 26, 19, 27. Since this path leads to the violation state of the TA in Figure 15 (specifically, a transition to the violation state is taken after transitions based on nodes 9, 21, and 2 are taken), FLAVERS will ignore this infeasible path during state propagation.

Note that, as described here, the definition of TAs is different from that of property automata and VAs in that the transitions in the TAs are based on the identities of TFG nodes instead of the labels of these nodes. Conceptually, we can label the TFG nodes with these unique node identifiers to be able to use the previously defined FSA model. In our implementation of FLAVERS, we redefine all FSA transitions in terms of TFG node identifiers for efficiency.

Construction of a TA requires $\mathcal{O}(|N_i|^2)$ steps where $N_i \in N$ is the set of nodes in T_i. In the worst case, the sub-flow-graph for the task is fully-connected, therefore we need to construct N_i transitions for each of the N_i states of the TA. Thus the entire algorithm is $\mathcal{O}(|N_i|^2)$. Constructing TAs is very fast in practice.

6.3 Combining the Property and Constraints in the State-propagation Analysis

To incorporate constraints into FLAVERS analyses, the state-propagation analysis is solved in conjunction with the analyses for each constraint. One obvious way to combine flow analyses for the property and all constraints is to form the product of the property and constraint automata, unifying all of the violation states, and then propagate states of this product automaton through the TFG. Here, we describe a more efficient approach that forms reachable product

automata states on-the-fly [NCO98].

Let $P = (\Sigma^P, S^P, \delta^P, A^P, s^P, s^P_{trap})$ be the property automaton and $\forall i, 1 \leq i \leq k, C_i = (\Sigma^{C_i}, S^{C_i}, \delta^{C_i}, A^{C_i}, s^{C_i}, v^{C_i})$ be the constraint automata used by the analysis. To handle these automata simultaneously, we define tuples of length $k+1$ that are associated with TFG nodes and represent the values of the property and constraints in these nodes. Formally, a *tuple* T is a collection of one state from each automaton in the problem:

$$T = (t^P, t^{C_1}, t^{C_2}, ..., t^{C_k}), \text{ where } t^P \in S^P \text{ and } \forall i : 1 \leq i \leq k, t^{C_i} \in S^{C_i}$$

Let *Tuples* be the set of all possible tuples:

$$\textit{Tuples} = \left\{ (t^P, t^{C_1}, t^{C_2}, \ldots t^{C_k}) \mid t^P \in S^P \wedge \bigwedge_{i=1}^{k} (t^{C_i} \in S^{C_i}) \right\}$$

The *initial tuple* is the tuple $T_0 = (s^P, s^{C_1}, s^{C_2}, ..., s^{C_k})$.

We associate a function f^n over tuples with each TFG node n:

$$\forall T = (t^P, t^{C_1}, t^{C_2}, ..., t^{C_k}) \in \textit{Tuples} : f^n(T) = (\overline{t^P}, \overline{t^{C_1}}, \overline{t^{C_2}}, \ldots, \overline{t^{C_k}}),$$

where

$$\overline{t^P} = \begin{cases} \delta^P(t^P, \textit{label}(n)) & \text{if } \textit{label}(n) \in \Sigma^P \\ t^P & \text{if } \textit{label}(n) \notin \Sigma^P \end{cases}$$

$$\forall 1 \leq i \leq k, \overline{t^{C_k}} = \begin{cases} \delta^{C_k}(t^{C_k}, \textit{label}(n)) & \text{if } \textit{label}(n) \in \Sigma^{C_k} \\ t^{C_k} & \text{if } \textit{label}(n) \notin \Sigma^{C_k} \end{cases}$$

As in the state-propagation algorithm for property automata in Section 5, we generalize f^n to a function F^n over sets of tuples for each TFG node, making sure that tuples containing violation states of constraints are not produced:

$$\forall n \in N, \forall X \subseteq \textit{Tuples}, F^n(X) = \left\{ f^n(T) = f^n(t^P, t^{C_1}, t^{C_2}, \ldots t^{C_k}) \mid T \in X \wedge \bigwedge_{i=1}^{k} (t^{C_i} \neq v^{C_i}) \right\}$$

The lattice elements for this data-flow problem are sets of tuples and the join operation is set union. Once the solution of the data-flow problem over the TFG converges to a fixed point, we look only at the final node of the TFG to determine whether the property holds. For each tuple associated with $n_{\textit{final}}$ we check the constraint automata states to see whether all of these are accepting states. If any constraint automaton is left in a non-accepting state, we remove the entire tuple from $n_{\textit{final}}$. We then look at the property states on the remaining tuples as was done in the state-propagation algorithm in Figure 10.

This tuple-based propagation approach is reminiscent of K-tuple frameworks from [MMR95]. An important distinction is that each component of a tuple in K-tuple frameworks corresponds to a special kind of edge in the graph. In our approach, an event associated with a TFG node can be present in alphabets of several automata and thus components of tuples are not directly tied to disjoint sets of information in the flow graph.

The proof of conservativeness of state-propagation analysis using constraints appears in Appendix A.3.

Theorem 7 (State-propagation Complexity). *Given a TFG G with nodes N^G, a property automaton P with states S^P, and constraint automata $C_1, \ldots, C_k$ with states $S^{C_1}, \ldots, S^{C_k}$, state propagation using the tuple based approach has worst-case complexity that is*
$\mathcal{O}(|S^P||S^{C_1}| \cdots |S^{C_k}||N^G|^2)$.

Proof. This is proved in [NCO98]. Intuitively, if the property automaton and constraint automata are treated as FSAs, their cross-product could be computed, yielding a single FSA with at most $|S^P| \cdot |S^{C_1}| \cdots |S^{C_k}|$ states. This FSA could be modified into a property automaton and used in the state-propagation algorithm from 5, with worst-case complexity $\mathcal{O}(|S^P||S^{C_1}| \cdots |S^{C_k}||N^G|^2)$. □

Using the VA for `count` and the TA for the task `ATM`, FLAVERS will still report that the property does not hold and present the analyst with a counterexample path. For inconclusive results there are usually several counterexamples that could be shown and the order that these are generated depends on how the worklist is implemented in the state-propagation algorithm [CCO01]. For instance, FLAVERS might return the path 20, 1, 9, 21, 2, 11, 22, 4, 14b, 23, 2, 18, 11, 22, 4, 14b, 23, 2, 18, 11, 22, 4, 14a, 26, 8, 19, 27. This path corresponds to the event sequence `count=0`, `is_count!=2`, `invalid`, `count_inc`, `is_count!=2`, `invalid`, `count_inc`, `is_count=2`, `eat_card` and indeed is an example of an executable path that violates the property. For this example, the analyst would soon conclude that the property shown in Figure 11 does not hold on this program because of an off-by-one error. In the program, the variable `count` is initialized to zero and the card is eaten when the variable is equal to two, which occurs after an invalid PIN has been entered three times. By changing the if statement that corresponds to CFG node 14 in Figure 5 to read `if(count=1)`, then state propagation, using the VA for `count` and the TA for the task `ATM`, reports that the property in Figure 11 holds.

Alternatively, the counterexample path 20, 9, 21, 11, 22, 14b, 23, 18, 11, 22, 14b, 23, 18, 11, 22, 14a, 26, 19, 27 might be generated. This path results in the same event sequence, but is infeasible because it does not follow the control flow of task `Customer`. Often when observing an infeasible counterexample, the analyst's attention is directed close enough to the source of the problem that the fault is still discovered. When this is not the case, the source of the infeasibility can be used to help select a constraint that will eliminate this infeasible path from further consideration. For the infeasible counterexample given above, the analyst would add a TA for task `Customer`.

7 Evaluation of FLAVERS

The complexity bounds of FLAVERS are polynomial in the number of program statements but exponential in the number of constraints. These are worst-case bounds, however, and do not indicate the expected cost of analysis on typical applications. To develop an understanding of the feasibility of FLAVERS, we applied it to the analysis of a variety of problems, where a problem consists of a program, property, and set of constraints. In this section, we describe our methodology for applying FLAVERS to a set of problems, present the results of our empirical evaluation, and discuss a number of observations.

7.1 Methodology

Our empirical evaluation was performed using an implementation of FLAVERS that is targeted for Ada tasking programs. The FLAVERS/Ada toolset has been programmed in three different languages. The tools that construct CFGs from the Ada source code of the program being analyzed are built on top of Arcadia infrastructure components [TBC+88] and are written in Ada. Tools written in Java convert these CFGs into a TFG and construct the constraint and property automata that are needed. Finally, the MHP and state-propagation analyses are written in C for efficiency.

We evaluated four scalable Ada tasking programs. Three of these, the Readers/Writers, Gas Station, and several variants of the Dining Philosopher program, have been used frequently as examples in the concurrency analysis literature (e.g., [ABC+91, Cha96, CA94, DBDS93, Dwy95, MR93, YTL+95]). They were selected because they represent variations on topologies and synchronization structures that appear to be commonly used in concurrent programs. Readers/Writers and Gas Station represent variations on centralized resource management strategies, and the different Dining Philosopher programs represent different approaches to distributed resource management strategies. In addition, for each of these programs, correctness properties are easily identified. The fourth example, the Chiron program, is derived from a real system [KCTT91] and has been used as an example for comparing finite-state verification approaches [ACD+99]. For the Chiron program, we developed several of our own properties by reverse engineering the program. The goal was to come up with properties that developers would actually want to check for this program.

Since the version of FLAVERS/Ada that we used is designed to work with programs that terminate, the code for some of these programs needed to be modified slightly to ensure this. The source code for all these programs can be found at `http://laser.cs.umass.edu/verification-examples/`.

These programs can be scaled by replicating some of the tasks. As is typically the case with finite-state verification

techniques, properties involving such tasks are stated in terms of only some of these tasks, such as customer 1 and customer 2. Informal arguments are then made that if the property holds for an arbitrary selection of such tasks, it would hold for any such selection.

There are two important concerns in this evaluation: cost and precision. For evaluating the former, we consider the cost of constructing the analysis artifacts as well as the cost of performing the analysis. To isolate concerns about cost from concerns about precision, we chose properties that are known to be valid for the selected programs and measured only the performance of the analyses that yielded conclusive results.

For evaluating precision, we report on the constraints that were included to achieve a conclusive result. Based on our assessment of the reason for an inconclusive result, we selected TA and VA constraints. This resulted in a minimal configuration, in the sense that removing one of the constraints from this configuration would result in an inconclusive analysis result. The resulting configuration is not necessarily an optimal solution, where optimal means that it took the least amount of execution time (or space) to produce conclusive results. A more rigorous approach would be needed to determine the optimal configuration, since adding a constraint may reduce the execution time (or space); so even after a conclusive configuration is found, additional constraints would need to be considered to determine an optimal configuration. Our goal was not to find the best performance but to determine if there is a reasonable process that analysts might follow to find a minimal configuration. In following such a process, analysts would most likely not add additional constraints after finding a set that produced conclusive results.

To gain a sense of how the analysis would scale as the size of the system increases, we attempted to scale programs up to 200 replications. Note that each replication may involve adding one or more tasks to the system. For example, for the dining philosopher program both a philosopher and a fork must be added for each replication. Before scaling a program, we first considered a small, yet reasonable, instantiation of that program and then found a minimal configuration of constraints that led to conclusive results. We then used this configuration as we scaled the size of the program. Of course, there is no guarantee that this configuration will continue to produce conclusive results for a property (or the correspondingly scaled version of a property) when a program is scaled to a larger size. This was the case, however, for almost all the programs and properties that we considered; exceptions are noted below.

Although these experiments give the cost of running FLAVERS on a system with a minimal set of constraints, they do not give any indication of the analyst's time, including the time to develop the properties, to examine counterexamples to determine if they are feasible or infeasible, and to determine which constraints should be added to remove infeasible paths from consideration. Such an evaluation is beyond the scope of this paper, but we provide anecdotal evidence when discussing the results of the evaluation.

Our primary measure of analysis time is the sum of user and system time as measured by `/usr/bin/time` on a Sun Enterprise 3500 with two 336 MHz processors and 2 GB of memory, running Solaris 2.6. Although this is a multi-user system, for all experiments, we had exclusive access to the machine to reduce variance in the times. The Ada components of the FLAVERS/Ada tools were compiled using the Verdix Ada Development System version 6.2.3c with optimizations disabled (to avoid known compiler bugs). The Java components were run using the Sun JDK version 1.3.0, with HotSpot. The C components were compiled with the Free Software Foundation's gcc version 2.95.2, using the -O2 flag for optimization.

When computing the execution time, we do not include the time for performing language processing, which would include syntactic and semantic analysis and the creation of an abstract syntax tree and annotated CFG. In our toolset, language processing is built upon an obsolete front end and is not representative of expected front-end costs. Moreover, these costs are well-understood and not of interest here. We include the cost of TFG construction, alphabet refinement, partial-order reduction, and state propagation.

To estimate the actual functional dependence between execution time and the size of the program, we fit both polynomial and exponential curves to the timing data. To evaluate the fit, we look at the mean square residuals, MS_{Res}, which is the amount of variability *not* explained by the model per degree of freedom. When comparing fits, a smaller value for MS_{Res} represents a better fit. We also give the mean square regression, MS_{Reg}, which is the amount of variability explained by the model per degree of freedom. MS_{Reg} is used to show the difference in scale between it and MS_{Res}. In our examples, MS_{Reg} is often orders of magnitude larger than MS_{Res}, indicating that the curves we are fitting to the data explain a large amount of the variance in the data. We supplement the fit data with plots that compare the fit data with the actual data.

To evaluate the space costs, we consider the size of the analysis artifacts compared to the size of the program. In our

evaluation LOC gives the number of lines of code for the Ada program, not counting blank lines and lines that contain only comments; Nodes and Edges are the number of nodes and edges of the TFG; Tuples is the number of different tuples created during state propagation; and Node-Tuples is the sum, over all nodes in the TFG, of the number of tuples associated with each node. Since the number of Node-Tuples corresponds to the number of states FLAVERS explored while performing state propagation, determining how the number of Node-Tuples grows with respect to program size gives a good indication of how memory usage by FLAVERS scales with respect to program size. To estimate the functional dependence between space and program size, we fit curves to the different size measure data.

The evaluation described here provides valuable insight into the FLAVERS approach. Since the sample size is small and not known to be representative, however, we do not claim that the results can be generalized to all Ada tasking programs.

7.2 Detailed Empirical Results

In this subsection, we describe each program and then discuss each property considered, the constraints used to produce conclusive results, and any interesting issues that arose. For each program, we then evaluate the performance of FLAVERS. The complete size and timing data for these evaluations are presented in Appendix B and Appendix C, respectively.

7.2.1 The Readers/Writers Program

The readers/writers program is a standard example that implements a means of thread-safe access to shared data. This program consists of a central data server, called the controller task, and a collection of reader and writer client tasks. It is scalable in the number of client tasks. In the program, readers only attempt to read and writers only attempt to write the shared data; the controller, however, can support clients that both read and write. The controller enforces exclusive-write semantics; if a writer is active then no other writer or reader can be active. These semantics are encoded using local variables `WriterPresent` and `ActiveReaders`.

The state space of this program grows exponentially with the number of client tasks, where a state of the program records the states of the controller, readers, and writers. To evaluate how the cost of the FLAVERS analysis scales for this program, we began with two readers and two writers and then increased the size of the program up to 200 readers and 200 writers. Assuming that a client task can only be in one of two states, there are 2^{r+w} reachable states for this program, where r is the number of readers and w is the number of writers. Thus, if we can demonstrate polynomial growth for accurate analysis, we will have improved on naïve reachability analysis.

We specify properties in terms of the events `rw.control.start_write`, `rw.control.stop_write`, `rw.control.start_read` and `rw.control.stop_read`, which are the fully-qualified Ada names of the controller entries, corresponding to a writer task starting and stopping writing and a reader task starting and stopping reading respectively. We use the abbreviations `wstart`, `wstop`, `rstart` and `rstop`, respectively, to make the QREs more readable.

Exclusive Read Write Property: The Exclusive Read Write Property checks that when a writer is active, no other reader or writer can become active. The QRE for this property is:

```
for events {wstart, wstop, rstart, rstop} show all executions satisfy
(~[wstart]*; wstart; ~[wstart, rstart, wstop]*; wstop)*;
(wstart; ~[wstart, rstart, wstop]*)?
```

The first line of the regular expression portion of the QRE states that between the time a writer starts writing and the time it stops, no other writer or reader can start. Note that in this property we explicitly list all the events that can not occur after a writer starts, instead of just listing the one event that could occur, namely `rstop`. We prefer this style since the regular expression portion does not have to be changed if additional events are added to the property alphabet, as often happens when adding VA constraints. The second line represents the case where a writer starts writing and does not stop. Although this is behavior that would be unexpected in the example, we are trying to capture the most general form of the property. This property was proved conclusively using two constraints: a TA for the controller and

a VA for the Boolean `WriterPresent` variable that keeps track of whether a writer task is currently writing. We used FLAVERS to prove this property for versions of this example with up to 200 readers and 200 writers; the analysis of the largest version required 29 seconds.

No Write While Reading Property: Although the **exclusive read write** property excludes the possibility of a write or a read being initiated while a writer is active, it does not preclude a reader from being active when a writer starts. A separate property can be used to check for this. Since multiple readers can be active simultaneously, this property needs to count the number of active readers. As a result, this property needs to be scaled with the number of readers. For the 2 and 3 readers case, the QREs for this property are:

```
for events {wstart, rstart, rstop}
show all executions satisfy
(
  (rstart;
    (rstart; rstop)*;
  rstop)*;
wstart*)*
```

2 readers

```
for events {wstart, rstart, rstop}
show all executions satisfy
(
  (rstart;
    (rstart;
      (rstart; rstop)*;
    rstop)*;
  rstop)*;
wstart*)*
```

3 readers

This QRE keeps track of the number of readers that have started but not yet stopped and ensures that a writer can only start while there are no active readers. To prove this property conclusively, FLAVERS needed 2 constraints, a TA for the controller and a VA for the integer counter variable `ActiveReaders` that keeps track of the number of active readers. Since the variable `ActiveReaders` can grow up to the number of readers in the example, the size of this automaton needs to be scaled with the size of the program. FLAVERS was able to prove that this property holds with 200 readers and 200 writers, but it required 6,901 seconds, a significant increase over the cost of proving the **exclusive read write** property, which did not require the property or constraints to scale as the program size was scaled.

We can take advantage of variable `ActiveReaders` to derive the following alternative expression of the property, **no write while reading 2**. The QRE for this property ensures that a write can only start when the value of the variable `ActiveReaders` is zero:

```
for events {r=0, is_r=0, r_inc, r_dec, wstart} show all executions satisfy
(~[wstart, r=0, is_r=0] | ([r=0, is_r=0]; ([r=0, is_r=0] | wstart)*;
  ~[wstart, r=0, is_r=0]))*; ([r=0, is_r=0]; ([r=0, is_r=0] | wstart)*)?
```

In this QRE, `r=0` represents an assignment of 0 to the variable `ActiveReaders`, `is_r=0` represents a check that the value is 0, `r_inc` and `r_dec` represent the incrementing and decrementing of this variable. In the program, these are the only events that access or change the value of `ActiveReaders`. The property checks that either `r=0` or `is_r=0` must precede a writer starting without an intervening change in the variable's value. This property does consider some event sequences as violations that are legal, for example, `r=0`, `r_inc`, `r_dec`, `wstart`. Still, if this property in conclusive, then we can conclude that no writer can start while a reader is reading. To prove this property conclusively, FLAVERS only needed a TA for the controller and proved this property on a example with 200 readers and 200 writers in about 30 seconds. Figure 16 shows the timing comparisons of the two formulations of this property.

Write First Property: Another property we consider is that data must be in the buffer before a read is attempted. The QRE for this property, **write first**, is:

```
for events {wstart, rstart} show all executions satisfy
(~[wstart, rstart]*; wstart; .*) | ~[rstart]*
```

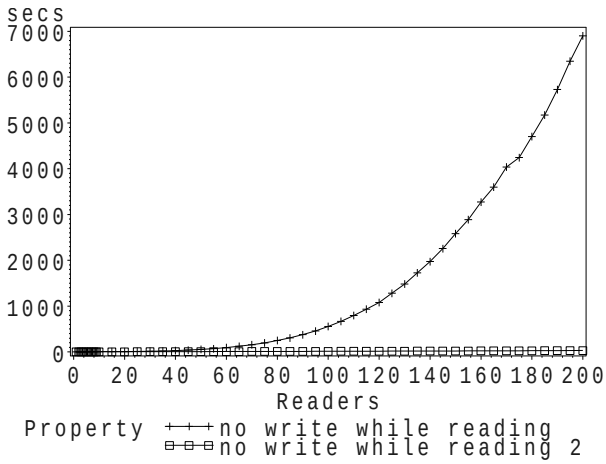

Figure 16: **no write while reading** Comparison

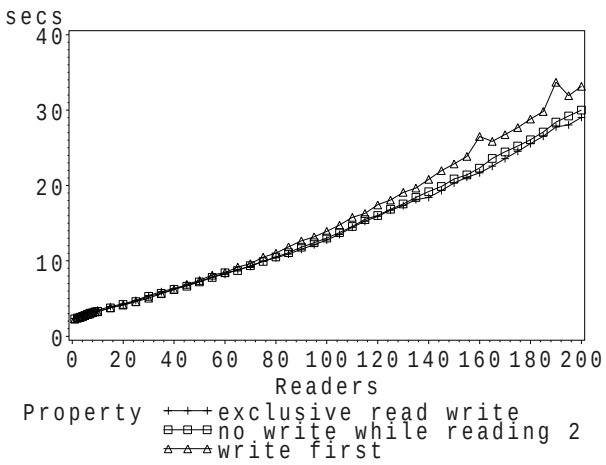

Figure 17: Reader/Writer Total Times

	$c_1n + c_0$	$c_2n^2 + c_1n + c_0$	$c_3n^3 + c_2x^2 + c_1n + c_0$	$e^{c_0+c_1n}$
MS_{Res}	$8.9763 * 10^5$	$4.7369 * 10^4$	$1.5484 * 10^3$	$3.2612{*}10^4$
MS_{Reg}	$1.3452 * 10^8$	$8.6837 * 10^7$	$5.8579 * 10^7$	$1.3043{*}10^8$
c_0	$-9.1272 * 10^2$	$2.7825 * 10^2$	-6.3646	4.3524
c_1	$2.6152 * 10^1$	$-2.4050 * 10^1$	2.4618	0.0227
c_2		$2.7208 * 10^{-1}$	$-9.3607 * 10^{-2}$	
c_3			$1.2660 * 10^{-3}$	

Table 1: Curve Fitting for **no write while reading**

The first clause of the regular expression ensures that a reader does not start until after a writer starts. The second clause says that if no writer starts throughout a program run, then no reader can start on this run. To prove this property conclusively FLAVERS used a TA for the controller and was able to handle an example with 200 readers and 200 writers in about 33 seconds.

Performance: Figure 17 shows execution time versus the number of readers in the example for all of the properties except for **no write while reading**, which was shown in Figure 16.

To estimate the actual functional dependence between execution time and the number of readers and writers, we fit different curves to the time needed to verify the four properties. Since **no write while reading** required significantly

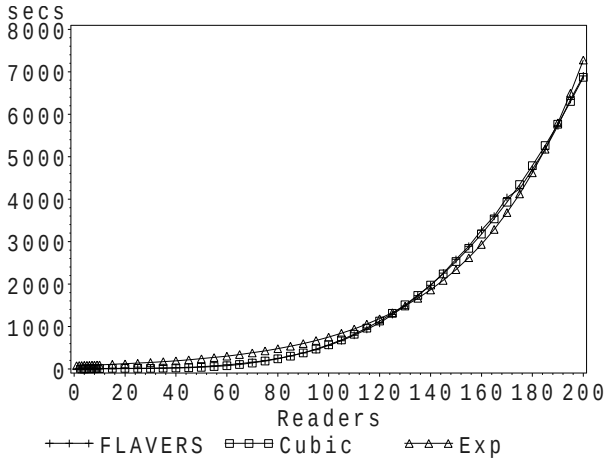

Figure 18: Fit Comparison for **no write while reading**

Readers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	75	25	50	16	47
50	1,153	711	1,912	7,905	23,959
100	2,253	1,411	3,812	30,805	92,909
150	3,353	2,111	5,712	68,705	206,859
200	4,453	2,811	7,612	121,605	365,809
n	$22n + 53$	$14n + 11$	$38n + 12$	$3n^2 + 8n + 5$	$9n^2 + 29n + 9$

Table 2: Size information for **no write while reading**

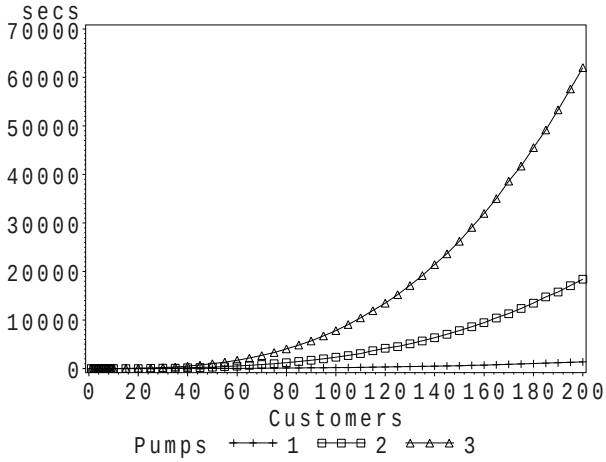

Figure 19: **pump 1 start/stop** Times

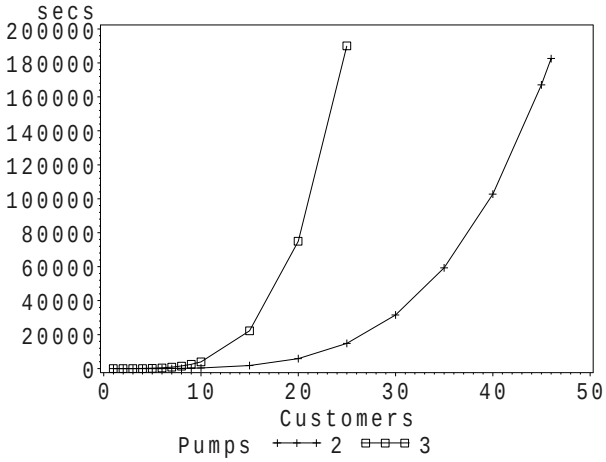

Figure 20: **correct change** Times

more time than the others, we only provide data for this property. The results of these fittings are shown in Table 1. Each column gives the data for the coefficients of the best-fit curve of the given form[5]. As is expected, for the three polynomial fits MS_{Res} decreases as the degree of the polynomial increases. Note, MS_{Res} for the exponential line is larger than MS_{Res} for the cubic polynomial. When comparing fits, a smaller value for MS_{Res} represents a better fit. One thing that is important to notice in this and all future tables is that the mean squared residuals are orders of magnitude smaller than the mean squared regression, meaning the curves we are attempting to fit to the data do a good job of explaining the data. Figure 18 plots the cubic polynomial against the exponential, and the cubic fit looks quite accurate, as expected from the fit data.

For property **no write while reading**, Table 2 gives the size information for some of the artifacts created during the analysis, for a sampling of program sizes. For this property, the numbers in these columns fit exactly to a polynomial, as shown in the last row of Table 2. The growth of the number of lines of code, nodes, and edges is linear in the number of readers, whereas the growth of the number of tuples and node-tuples is quadratic. The fact that the number of node-tuples grows quadratically is particularly important because the number of node-tuples corresponds closely with the amount of memory used for running an analysis. Thus, FLAVERS is showing polynomial growth both in terms of space and time on this property.

7.2.2 The Gas Station Program

The gas station program is a simulation of an automated self-serve gas station [HL85]. The gas station consists of a collection of server tasks, or pumps, an operator task, and a collection of client tasks, or customers. It is similar to the readers/writers program in that both have a server with client tasks. In the readers/writers program, the server is the control task. In the gas station program, the server is a scalable collection of cooperating tasks. Thus, the gas station program is scalable in two different dimensions, size of the server and number of clients.

[5]We also looked at curves of the form $e^{c_1 x}$, but in all cases this curve was a very poor fit so we do not include it here.

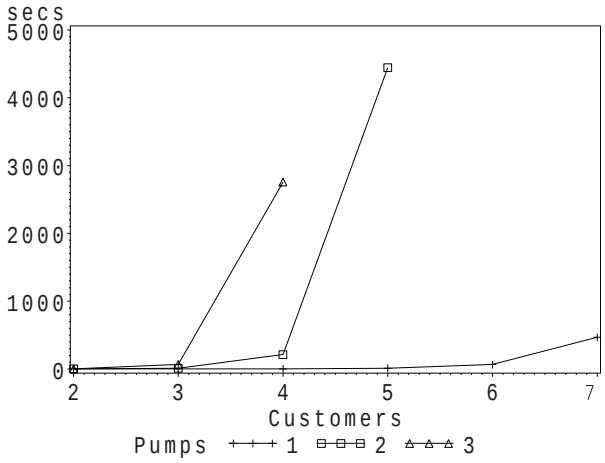

Figure 21: **mutual exclusion** Times

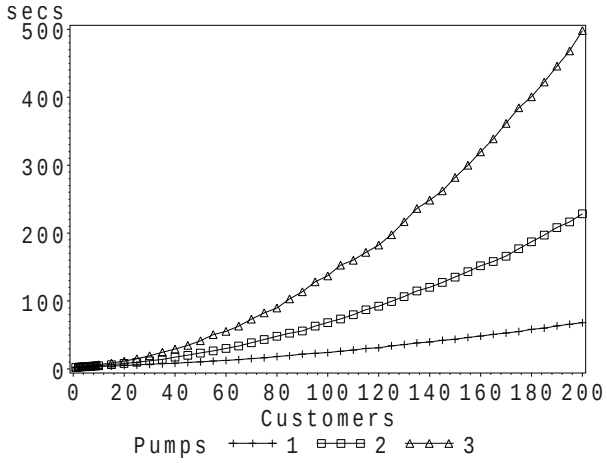

Figure 22: **customer 1 start/stop** Times

The server subsystem consists of an operator task that accepts payments and gives change to customers and a number of pump tasks that independently start and stop the pumping of gas. The operator interacts with the pumps by enabling them to pump gas after payment has been received and by getting information about how much gas was pumped. The client tasks pay for gas, pump it, and get their change. Instead of an explicit queue to represent the clients that are waiting to pump gas, the customers block on a rendezvous until the pump is available.

We looked at programs with 1, 2, and 3 pumps and attempted to scale the number of customers up to 200 for all properties. We did not look at systems with more than 4 pumps because, as mentioned, the language processing tools are built on an obsolete front end and the cost for running them became onerous.

Mutual Exclusion Property: The gas station program should only allow one customer to use a pump at a time. We check this property for customers 1 and 2 on pump 1. Since all the customer and pump tasks are identical and none reference task ids, we argue that verifying this property for these specific customers and pump is equivalent to verifying it for any arbitrary pair of customers and any particular pump. This property is specified in terms of the events indicating that customers 1 and 2 start and stop pumping using pump 1: `cust_1_start_pump_1`, `cust_1_stop_pump_1`, `cust_2_start_pump_1`, and `cust_2_stop_pump_1`. We abbreviate these by `c1_start`, `c1_stop`, `c2_start`, and `c2_stop` to make the QREs more readable. The QRE for the **mutual exclusion** property is:

```
for events {c1_start, c1_stop, c2_start, c2_stop}
show all executions satisfy
( (c1_start ; c1_stop) | (c2_start ; c2_stop) )*
```

This QRE ensures that either customer 1 starts pumping and then stops pumping or customer 2 starts pumping and then stops pumping. To prove this property, FLAVERS needs a TA for pump 1 and for each customer. Thus, there are $c + 1$ constraints, where c is the number of customers. Because of the number of constraints, the analysis of this property scaled very poorly. As shown in Figure 21, FLAVERS could only prove this property on an example with 1 pump and 7 customers, with 2 pumps and 5 customers, and with 3 pumps and 4 customers; larger sizes ran out of memory.

As with the reader-writer program, we looked for an alternative way to express this property. By focusing just on an individual customer and pump, say customer 1 and pump 1, we can create two simpler properties **customer 1 start/stop** and **pump 1 start/stop** that can be used together to prove the original **mutual exclusion** property.

The **customer 1 start/stop** property states that customer 1 starts pumping and then stops pumping. The QRE for this is:

```
for events {c1_start, c1_stop} show all executions satisfy
(c1_start ; c1_stop)*
```

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	197	76	140	187	567
2	316	131	738	2,638	30,946
5	673	296	5,268	25,117	1,395,485
10	1,268	571	21,938	112,892	12,494,400
15	1,863	846	50,008	263,667	43,692,665
20	2,458	1,121	89,478	477,442	105,385,280
25	3,053	1,396	140,348	754,217	207,967,245
$n > 2$	$119n + 78$	$55n + 21$	$228n^2 - 86n - 2$	$1260n^2 - 1345n + 342$	$13860n^3 - 13813n^2 + 1478n + 920$

Table 3: Size information for **correct change** with 3 Pumps

To prove this property conclusively, FLAVERS did not need any constraints. FLAVERS was able to prove this property with 200 customers and 3 pumps as shown in Figure 22. Since all customers tasks are identical, we can use the proof that customer 1 obeys the **customer 1 start/stop** property to argue that customer `i` also starts pumping and then stops pumping.

The QRE for the second property, **pump 1 start/stop**, is:

```
for events {c1_start, c1_stop, c2_start, c2_stop,..., cn_start, cn_stop}
show all executions satisfy
( [c1_start, c2_start,..., cn_start] ; [c1_stop, c2_stop,..., cn_stop] )*
```

This property states that a start event for some customer is always followed by a stop event for some customer, although not necessarily the same customer. To prove this property conclusively, FLAVERS needs a TA for pump 1. As shown in Figure 19, FLAVERS proved this property on the gas station example with 3 pumps and 200 customers.

Now we need to show that proving these two properties is sufficient to prove the mutual exclusion property. The alphabets of properties **customer 1 start/stop**, **pump 1 start/stop**, and **mutual exclusion** are not the same. Note, however, that for any property, we can add an event to its alphabet and not change its semantics by adding self-loop transitions for that new event to each of the property states. Let P_{ME} represent the property **mutual exclusion**, P_{C1} the property **customer 1 start/stop**, P_{P1} the property **pump 1 start/stop**, and P_{C2} the **customer 2 start/stop** property, all extended so that their alphabets are the same.

Now, let ρ be a path that represents a feasible execution of the gas station program. Since P_{C1}, P_{C2}, and P_{P1} were all proven conclusively, then $\rho \in \mathcal{L}(P_{C1})$, $\rho \in \mathcal{L}(P_{C2})$, and $\rho \in \mathcal{L}(P_{P1})$. Therefore, $\rho \in \mathcal{L}(P_{C1}) \cap \mathcal{L}(P_{C2}) \cap \mathcal{L}(P_{P1})$. Now, it can be shown that $\mathcal{L}(P_{C1}) \cap \mathcal{L}(P_{C2}) \cap \mathcal{L}(P_{P1}) \subseteq \mathcal{L}(P_{ME})$. Thus, since every feasible path is contained in the language of P_{ME}, we can conclude that the mutual exclusion property holds for the gas station program.

Correct Change Property: Another desirable property is that if a customer prepays on a given pump, then that customer receives the change for that pump. The following QRE expresses this property for customer 1 and pump 1:

```
for events {c1_prepay_p1, c1_change_p1, c1_change_p2,..., c1_change_pn}
show all executions satisfy
( ( c1_prepay_p1 ; c1_change_p1 ) | [c1_change_p2,..., c1_change_pn] )*
```

This QRE ensures that if customer 1 prepays on pump 1, then customer 1 receives change on pump 1. To prove the property conclusively, FLAVERS needs TAs for the operator, customer 1, and pump 1. Figure 20 shows FLAVERS' performance on this property. In this case, FLAVERS was only able to prove the property with 2 pumps and 46 customers and with 3 pumps and 25 customers; with more customers FLAVERS ran out of memory. Table 3 shows the size information for this property with 3 pumps.

Performance Of the gas station properties where FLAVERS could handle programs that scale to 200 customers, it takes the most time to verify the **pump 1 start/stop** property, thus we show the curve fittings for this property.

	$c_1n + c_0$	$c_2n^2 + c_1n + c_0$	$c_3n^3 + c_2n^2 + c_1n + c_0$	$e^{c_0+c_1n}$
MS_{Res}	$5.9292 * 10^7$	$1.6652 * 10^6$	$5.4350 * 10^3$	$3.1968*10^6$
MS_{Reg}	$1.2483 * 10^{10}$	$7.5678 * 10^9$	$5.0701 * 10^9$	$1.2010*10^{10}$
c_0	$-8.0699 * 10^3$	$1.7321 * 10^3$	$1.9707 * 10^1$	7.0802
c_1	$2.5193 * 10^2$	$-1.6125 * 10^2$	-1.7397	0.0201
c_2		2.2393	$3.9162 * 10^{-2}$	
c_3			$7.6166 * 10^{-3}$	

Table 4: Curve Fitting for **pump 1 start/stop** with 3 Pumps

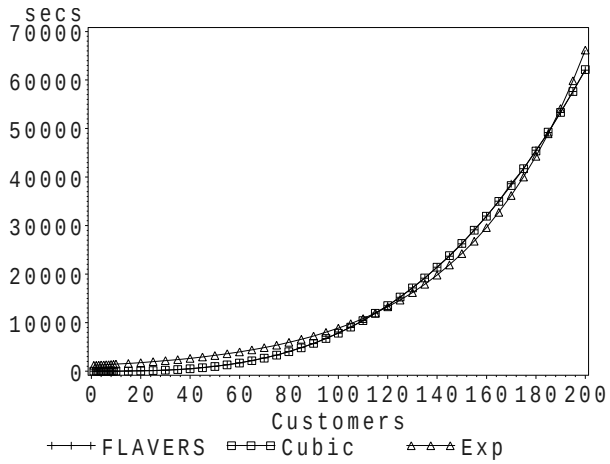

Figure 23: Fit Comparison for **pump 1 start/stop** with 3 Pumps

Table 4 shows the results of these fittings. With this property, MS_{Res} is better for the exponential curve than the linear polynomial, but the cubic fit is best, as can also be seen in the plot shown in Figure 23.

Table 5 gives size information for a sampling of program sizes. For this property, these sizes fit exactly to a polynomial, as shown in the last row of the table, when the number of customers is 2 or more. For these problems, the number of LOC, nodes, and tuples grew linearly with the number of customers and the number of edges and node-tuples grew quadratically.

There were two gas station properties where FLAVERS was not able to scale up to 200 customers, the **mutual exclusion** property and the **correct change** property. On the **mutual exclusion** property, the number of constraints needed to be increased as the number of customers increased, resulting in FLAVERS running out of memory. Unfortunately, there are not enough data points for this property to do meaningful curve fitting. Still, by verifying three other properties, we were able to prove that the **mutual exclusion** property holds on a system with 200 customers. For the **correct change** property, the number of constraints did not need to be increased with the number of customers, but FLAVERS was still unable to verify this property on a system with 200 customers. In this case, the number of node-tuples grew as a cubic polynomial of the number of customers, but the cubic coefficient was large, so the memory requirements grew quickly. We are unsure as to why the number of node-tuples grew so quickly for the **correct change** property.

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	197	76	136	10	292
2	316	131	878	17	1,580
50	6,028	2,771	669,518	353	761,180
100	11,978	5,521	2,689,018	703	3,027,330
150	17,928	8,271	6,058,518	1,053	6,798,480
200	23,878	11,021	10,778,018	1,403	12,074,630
$n > 1$	$119n + 78$	$55n + 21$	$270n^2 - 110n + 18$	$7n + 3$	$301n^2 + 173n + 30$

Table 5: Size information for **pump 1 start/stop** with 3 Pumps

Example	**mutual exclusion**	**no fork 1 up twice**
Standard	TA: phil_1, phil_2, fork_2	TA: fork_1
Fork Manager	TA: manager, phil_1, phil_2; VA: fork_2	TA: manager
Dictionary	TA: phil_1, phil_2, fork_2	TA: fork_1
Host	TA: phil_1, phil_2, fork_2	TA: fork_1

Table 6: Constraints for the Dining Philosopher Programs

7.2.3 The Dining Philosophers Program

The dining philosophers program consists of equal numbers of philosophers and forks. These are organized into a ring with alternating philosopher and fork tasks. Each philosopher has access to two forks, left and right, and thus each fork is shared by two philosophers. A philosopher attempts to gain access to both of its incident forks. A philosopher that has both forks will proceed to eat. Between the time a philosopher puts down the forks and makes an attempt to pick up the forks again, the philosopher is thinking. This program has a potential deadlock where each philosopher has picked up the left (right) fork and is waiting for the right (left) fork to become available.

We verified properties for four different versions of the dining philosopher program: standard, dictionary, fork manager, and host. The standard version prevents the potential deadlock by having each philosopher pick up the left fork and then the right fork, except for one philosopher, who reverses this order. In the standard version, each philosopher and fork is a task, so a program with n philosophers has $2n$ tasks. The dictionary version adds a token, the dictionary, that prevents its holder from picking up a fork and so it also has $2n$ tasks when there are n philosophers. When the left neighbor is thinking, the philosopher can pass the dictionary to that neighbor. The fork manager version uses a manager task to control access to the forks and ensures that philosophers pick up both forks in an atomic action. In this version, the manager task controls access to all of the forks, and so there is only one fork task and thus $n + 1$ tasks when there are n philosphers. The host version uses a control task to ensure that at most $n - 1$ philosophers can be attempting to pick up the forks at the same time. In this version, each philosopher and each fork is a task and there is one additional task for the control task and so the host version with n philosophers has $2n+1$ tasks. We checked the dining philosopher properties described below on each version of the dining philosopher program, scaling the number of philosopher and fork tasks up to two hundred each.

Mutual Exclusion Property: To check the property that "adjacent philosophers cannot eat concurrently," we need to choose two representative adjacent philosophers, say philosophers 1 and 2. To check this property, we added annotations to the code to identify the places where each philosopher starts and stops eating[6]. In the QRE representation of the property, we use `p1start` and `p1stop` to represent the events of philosopher 1 starting eating and stopping eating and use `p2start` and `p2stop` for the same actions of philosopher 2. The QRE for the **mutual exclusion** property is:

```
for events {p1start, p1stop, p2start, p2stop} show all executions satisfy
( ( p1start ; p1stop ) | ( p2start ; p2stop ) )*
```

This QRE is similar to the mutual exclusion property for the gas station program, in that it ensures that once philosopher 1 (or 2) starts eating then philosopher 2 (or 1) cannot eat. FLAVERS is able to prove this property on all four versions of the dining philosopher program using the constraints shown in Table 6. The times necessary to verify this property are shown in Figure 24.

No Fork Up Twice Property: To check that a fork cannot be picked up twice in a row without first being put down, we again need to represent this property in terms of an arbitrary fork instance. The events `up` and `down` represent the fork being picked up and put down, respectively. The QRE to check this property on fork 1 is:

```
for events {up, down} show all executions satisfy
(~[up]*; up; ~[up,down]*; down)*; ~[up]*; (up; ~[up,down]*)?
```

[6]We did not have to add annotations for analysis of reader-writer and gas station examples because all properties for these examples used task rendezvous events, which are automatically identified by FLAVERS.

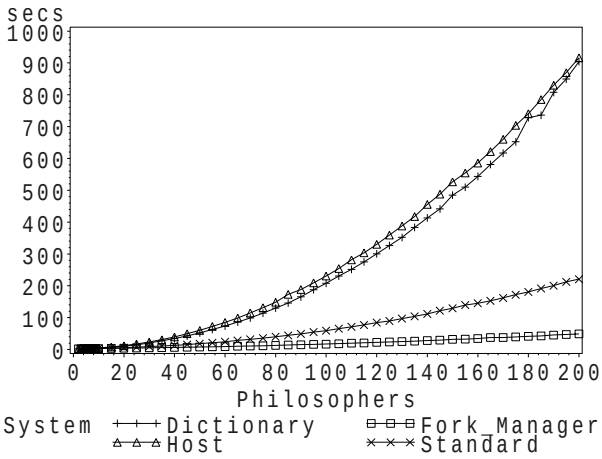

Figure 24: **mutual exclusion** Total Times

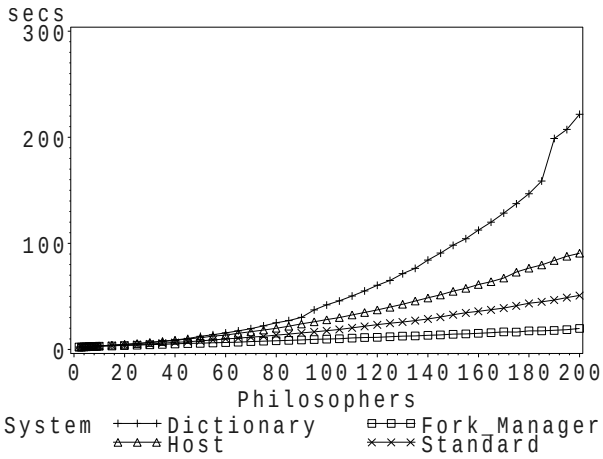

Figure 25: **no fork up twice** Times

	$c_1n + c_0$	$c_2n^2 + c_1n + c_0$	$c_3n^3 + c_2n^2 + c_1n + c_0$	$e^{c_0+c_1n}$
MS_{Res}	$5.8657 * 10^3$	5.038	5.069	$1.8156{*}10^3$
MS_{Reg}	$3.4409 * 10^6$	$1.8523 * 10^6$	$1.2349 * 10^6$	$3.5543{*}10^6$
c_0	$-1.0317 * 10^2$	2.2300	2.6553	3.8761
c_1	4.2639	$2.5399 * 10^{-2}$	$-1.1591 * 10^{-2}$	0.0152
c_2		$2.2734 * 10^{-2}$	$2.3237 * 10^{-2}$	
c_3			$-1.731 * 10^{-6}$	

Table 7: Curve Fitting for **mutual exclusion** on the Host Version

The first line of the regular expression part of this property ensures that once an `up` for fork 1 occurs, no `up`s will be seen until a `down` for this fork occurs. The second line deals with the case where the execution might terminate with an `up` that is not followed by a `down`. It ensures that two `up`s are not seen consecutively if this situation occurs. FLAVERS is able to prove this property on all four versions using the constraints shown in Table 6. Figure 25 shows FLAVERS' performance on this set of problems.

Performance Of the four versions of the dining philosopher program, FLAVERS takes the most time to verify the **mutual exclusion** property on the host version. Table 7 shows the results of curve fitting for timing results for this version. This time, the quadratic appears to be the best fit, as can be seen in the plot shown in Figure 26.

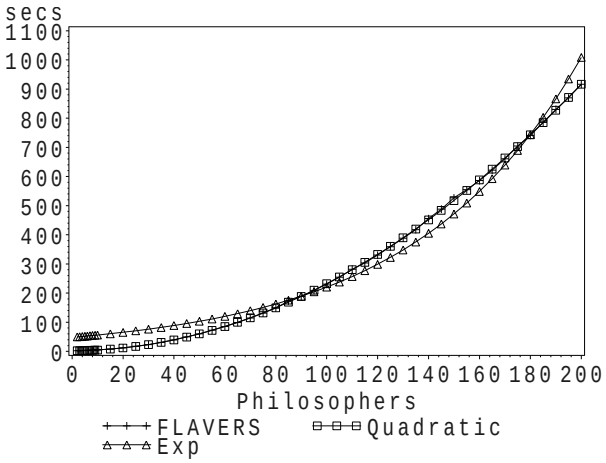

Figure 26: Fit Comparison for **mutual exclusion** on the Host Version

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
2	87	44	132	87	445
3	114	62	305	87	2,118
50	1,383	908	81,662	87	66,884
100	2,733	1,808	323,412	87	135,784
150	4,083	2,708	725,162	87	204,684
200	5,433	3,608	1,286,912	87	273,584
$n > 2$	$27n + 33$	$18n + 8$	$32n^2 + 35n - 88$	87	$1378n - 2016$

Table 8: Size information for **mutual exclusion** on the Host Version

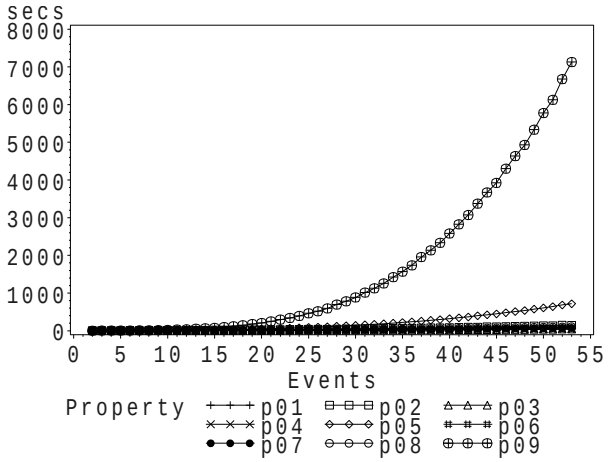

Figure 27: Chiron Total Times

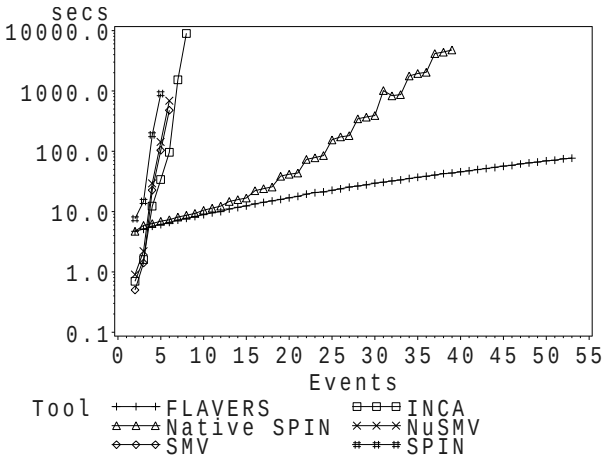

Figure 28: Tool Comparison for **p07** on original Chiron

Table 8 gives the size information for a sampling of program sizes. For the **mutual exclusion** property and a program with at least three philosophers or more, these sizes fit exactly to a polynomial, as shown in the last row of the table. For this property, the number of tuples was a constant; the number of LOC, edges, and node-tuples grew linearly with the number of philosophers, and the number of edges grew quadratically.

7.2.4 The Chiron Problem

Chiron is a user interface framework developed at the University of California at Irvine [KCTT91]. Since it was not developed as an analysis test case, it may be more representative of "typical" concurrent Ada programs than some of the other programs we considered.

In an instantiation of Chiron, there are one or more artist tasks that draw on the screen. There are also a number of events, for example button pushes, that may be of interest to artists. To listen for events, artists register with a central server called the dispatcher. The dispatcher maintains a list of artists registered for each event, and when it receives an event, it passes it on to all artists registered for that event. An alternative to a centralized dispatcher is to have dedicated dispatchers for each event. This alternative "decomposed dispatcher" version of Chiron was created and considered in our evaluation.

We checked properties of Chiron program with a centralized dispatcher and two artists, and scaled up the number of events for which artists could register. The two artists were not identical; `artist1` registered for one third of the events, `artist2` registered for another third, and both artists registered for the remaining one third of the events.

We checked the nine properties on this Chiron program. The description of the properties and the constraints needed to prove each are shown in Table 9. In Chiron, the dispatcher uses an array for each event to keep track of which artists are registered for that event. A VA called "event$i$ list size" keeps track of the number of spaces in the array that are used, that is, the number of artists that are registered for event i. A VA called "event$i$ list slot j" keeps track of the data that is stored in the jth position of the array that stores the artists that are registered for event i.

Property	Description	Constraints
p01	`artist1` never registers for `event1` if it is already registered for this event, and `artist1` never unregisters for `event1` if it is not registered for this event.	TA: artist1
p02	If `artist1` is registered for `event1` and the dispatcher receives `event1`, then the dispatcher will not accept another event before passing `event1` to `artist1`.	TA: dispatcher; VA: event1 list size, event1 list slot 1, event1 list slot 2
p03	The dispatcher does not notify any artists of `event1` until it receives an `event1`.	TA: dispatcher
p04	Having received `event1`, the dispatcher never notifies artists of `event2`.	TA: dispatcher
p05	If no artists are registered for `event1`, the dispatcher does not notify any artist of `event1`.	TA: artist1, artist2
p06	The dispatcher never gives `event1` to `artist1` if `artist1` is not registered for `event1`.	TA: artist1
p07	If `artist1` registers for `event1` before `artist2` does, then when the dispatcher receives `event1` it will first notify `artist1` and then `artist2` of this event.	TA: dispatcher; VA: event1 list size, event1 list slot 1
p08	The size of the list used to store the IDs of artists registered for `event1` never exceeds the number of existing artists.	TA: dispatcher; VA: event1 list size, event1 list slot 1, event1 list slot 2
p09	The program does not terminate while there is an artist that is registered for an event.	TA: artist1, artist2

Table 9: Chiron Properties

	$c_1n + c_0$	$c_2n^2 + c_1n + c_0$	$c_3n^3 + c_2n^2 + c_1n + c_0$	$e^{c_0+c_1n}$
MS_{Res}	$8.7971 * 10^5$	$4.8461 * 10^4$	$4.221 * 10^2$	$2.9107{*}10^4$
MS_{Reg}	$1.6812 * 10^8$	$1.0487 * 10^8$	$7.0696 * 10^7$	$1.7514{*}10^8$
c_0	$-1.6561 * 10^3$	$7.0291 * 10^2$	$-7.2032 * 10^1$	4.2851
c_1	$1.1981 * 10^2$	$-1.2453 * 10^2$	$2.5074 * 10^1$	0.0875
c_2		4.4426	-2.1808	
c_3			$8.0284 * 10^{-2}$	

Table 10: Curve Fitting for **p09**

Timing information for all of the Chiron properties are shown in Figure 27. Unfortunately, we could only scale the program up to 53 events, because of limitations with the language processing tools that we used. FLAVERS verified all of the nine properties, up to 53 events.

Performance Of the Chiron properties, it takes the most time to verify property **p09**, and thus, in Table 10, we show the curve fittings for only this property. For this property, MS_{Res} is best for the cubic polynomial. This can be seen in the plot shown in Figure 29.

Table 11 gives the size information for a sampling of program sizes. Unlike the previous properties, these sizes do not fit exactly into polynomials, even when some of the smaller sized programs are not considered. Thus, we performed curve fitting on the size metrics shown in Table 11. We found that the best-fit curve for the number of LOC and nodes is linear, for the number of edges and tuples it is quadratic, and for the number of node-tuples it is cubic. Since the amount of space used for an analysis is closely related to the number of node-tuples, we show the curve fitting data for this metric in Table 12 and Figure 30.

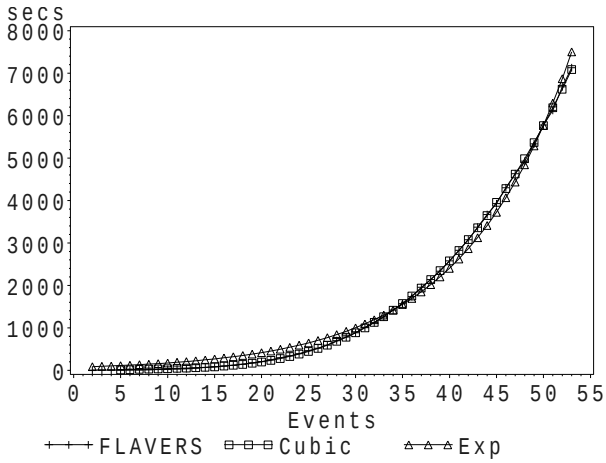

Figure 29: Fit Comparison for **p09**

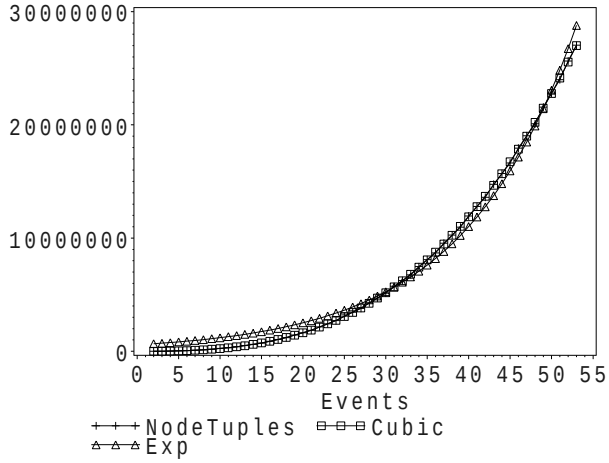

Figure 30: Fit Comparison for **p09** Node-Tuples

Events	LOC	Nodes	Edges	Tuples	Node-Tuples
2	556	96	374	559	5,984
10	2,393	314	2,876	9,101	250,542
20	4,690	582	8,894	33,451	1,641,572
30	6,987	850	17,782	73,126	5,132,994
40	9,286	1,124	30,476	130,001	11,957,892
50	11,583	1,392	46,134	200,751	22,799,552
53	12,274	1,473	51,442	225,214	27,030,782

Table 11: Size information for **p09**

7.2.5 Tool Comparison

The study reported in [ACD+99] used the Chiron program to compare different finite-state verification techniques across a range of properties and for different sizes of systems. The tools considered included both explicit state and symbolic model checkers, SPIN [Hol97] and SMV [McM93] respectively, an integer programming based analysis, INCA [Cor96], and FLAVERS.

To give a sense of the variation in performance, we show the comparative run-times of the tools on the original centralized dispatcher and the decomposed dispatcher versions of Chiron, with two artists and increasing numbers of events. For the original Chiron program, we show the comparison for property p07 in Figure 28, where FLAVERS exhibited its best comparative performance, and for property p09 in Figure 31, where FLAVERS exhibited its worst comparative performance. In both cases, FLAVERS appears to scale significantly better than the other verification techniques. The performance of different verification tools is highly-sensitive to program structure. Although FLAVERS performs equally well for both the centralized and decomposed dispatcher versions of Chiron, the performance of other tools improves, in some cases dramatically, for the decomposed dispatcher version as illustrated in Figure 32. Although FLAVERS does not have the best absolute run-time for that property, the rate of growth of its

	$c_1n + c_0$	$c_2n^2 + c_1n + c_0$	$c_3n^3 + c_2n^2 + c_1n + c_0$	$e^{c_0+c_1n}$
MS_{Res}	$9.1562 * 10^{12}$	$2.1581 * 10^{11}$	$2.7322 * 10^{9}$	$6.3260{*}10^{11}$
MS_{Reg}	$2.7918 * 10^{15}$	$1.6195 * 10^{15}$	$1.0832 * 10^{15}$	$3.0010{*}10^{15}$
c_0	$-6.1078 * 10^{6}$	$1.6260 * 10^{6}$	$-6.1071 * 10^{3}$	$1.3259{*}10^{1}$
c_1	$4.8821 * 10^{5}$	$-3.1284 * 10^{5}$	$2.2593 * 10^{3}$	0.0739
c_2		$1.4565 * 10^{4}$	$6.1458 * 10^{2}$	
c_3			$1.6909 * 10^{2}$	

Table 12: Curve Fitting for **p09** Node-Tuples

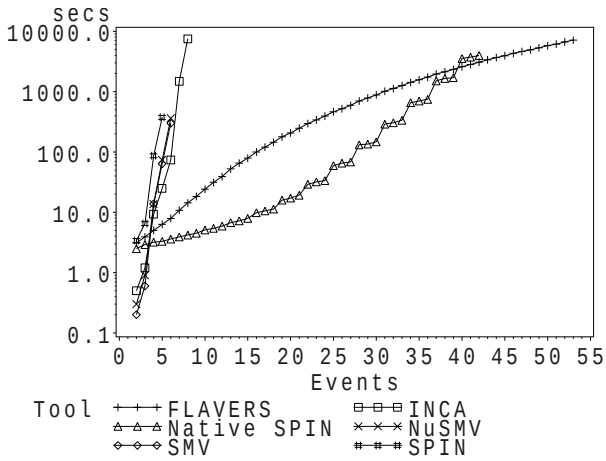

Figure 31: Tool Comparison for **p09** on original Chiron

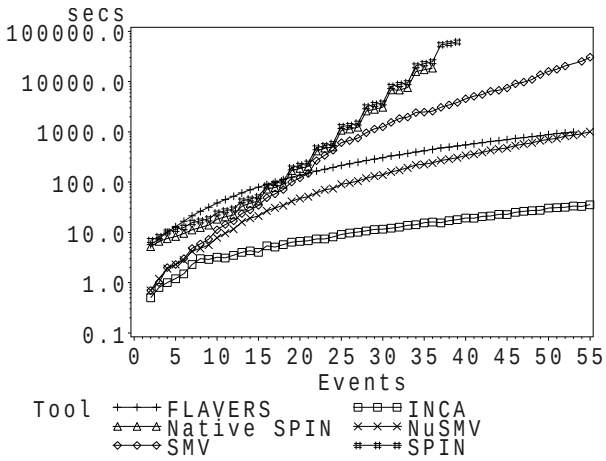

Figure 32: Tool Comparison for **p07** on decomposed dispatcher Chiron

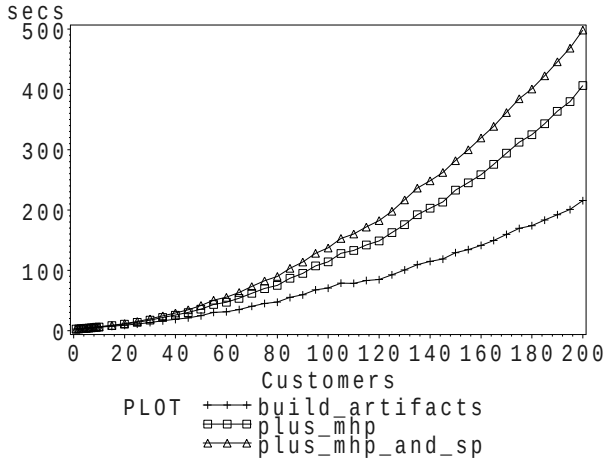

Figure 33: Cost Breakdown for **customer 1 start/stop** with 3 Pumps

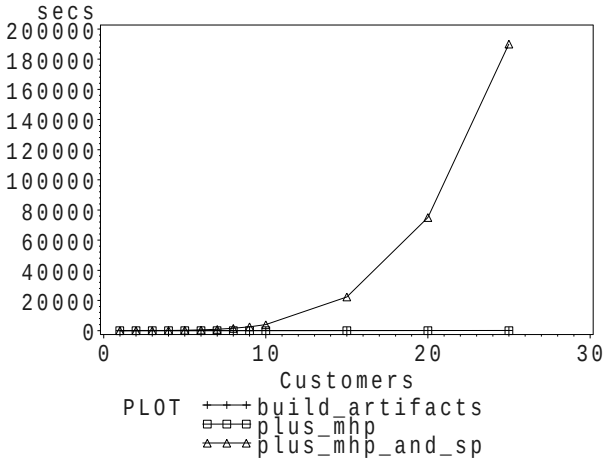

Figure 34: Cost Breakdown for **correct change** with 3 Pumps

run-time appears better than or at least quite competitive with the other tools.

7.3 Model construction

The times presented in the previous subsection included the cost for buiding the artifacts from the CFG and running state propagation. Time usually grew no worse than cubically in the size of the system being analyzed. To determine where the time was spent, we broke the process down into three steps: the cost for building the property automaton, the constraint automata, and the TFG without MIP edges; the cost for computing the MHP information and adding the MIP edges; and the cost for running state propagation.

Figures 33 and 34 show this breakdown for two of the properties of the Gas Station system. The lines in the plot show cumulative times for the process, meaning the lowest line is just the time for building the initial artifacts; the middle line includes both the time for building the artifacts, running the MHP algorithm, and adding the MIP edges; and the top line is the time for the entire process, including state propagation.

Across all of the properties presented in the paper, the plots tended to look like one of these two plots. Either the three times grew relatively evenly as the system size increased, as in Figure 33, which shows the time for the **customer**

1 start/stop property on the Gas Station with 3 pumps, or the cost of state propagation was the dominant cost for the process, as in Figure 34, which shows the time for the **correct change** property on the Gas Station with 3 pumps.

Since the cost for the overall process was usually no worse than cubic, this means that the cost for each of these steps was no worst than cubic.

7.4 Discussion

The empirical results in this section demonstrate the feasibility of FLAVERS analyses. In particular, they demonstrate that there exist relevant properties of non-trivial programs that can be efficiently verified using an analysis approach where the precision of the model is improved incrementally. It was shown empirically that the computational time and space for a FLAVERS analysis, starting from annotated CFGs through to finding a conclusive result once the set of constraints was known, grew at a rate that was usually no more than cubic.

In this study, we looked at three properties for the Readers/Writers program, two properties for the Gas Station program, two properties for each of four variants of the Dining Philosopher program, and nine properties for the Chiron program, for a total of 22 properties. Of these, 19 were scalable to programs up to 200 replications (or in the case of Chiron 53), where a replication may have required the addition of a single task, such as a customer in the Gas Station program, or several tasks, such as a philosopher and a fork in the standard Dining Philosopher program. Two of the properties, the **mutual exclusion** property of the Gas Station program and the **no write while reading** property of the Readers/Writers program, had to be reformulated before they could be verified on programs scaled to 200 replications. The property that could not be scaled in our experiments, the **correct change** property of the Gas Station program, appears to need space cubic in the number of customers in the system, but with a large coefficient in front of the cubic term, as shown in Table 3.

For all the programs we considered, both the lines of code and the number of tasks in the system grew linearly as we scaled the programs. While FLAVERS' worst-case performance is $\mathcal{O}(S^6)$, where S is the number of statements in the program, our experimental results provide good evidence that in practice we can expect to see performance that is $\mathcal{O}(S^3)$. In practical terms, there was only one property of the 22 properties that could not be scaled.

For most of the properties verified, the number of constraints needed did not change when the program size was scaled up. Exceptions were the **mutual exclusion** and **correct change** properties of the Gas Station program. For these properties, we could not scale the programs as far as we wanted. We also failed to scale the Chiron analyses as far as we wanted, but this was due to limitations in the language-processing tools.

Although the number of constraints remained constant for most problems, the size of some of the constraints needed to be scaled as the program size was scaled. For example, in the **correct change** property of the Gas Station program, the number of states in the TAs for the operator and pump tasks grows linearly with the size of the program. Although FLAVERS did not scale well on this problem, there were other problems where the size of the constraints scaled and where the analysis did scale well. The number of states in the TA also grows linearly with the number of philosophers for the manager task in the **mutual exclusion** property of the Dining Philosopher program with the Fork Manager. Verifying this property on the Fork Manager version required less time than on the Dictionary and Host versions, where the size of the constraints remained constant as the programs scaled.

For almost all of the scalable programs we considered, the rate of growth of analysis cost was best explained by a cubic polynomial with respect to the number of tasks in the program. This rate of growth included the time for generating all of the artifacts, starting from the CFGs. For this study, we do not consider LOC to be a good metric for measuring the complexity of the program. By using alphabet refinement, FLAVERS builds its model from only the portions of the program relevant to the property being checked. Thus, a large program with sparse events will have a small model. Still, with respect to the number of LOC in the system, the worst-case bound of FLAVERS is a sixth-degree polynomial. For all of the problems we evaluated, the number of LOC grew linearly with the number of tasks in the system. This means FLAVERS' actual performance was cubic with respect to the number of LOC on these system. Similarly, we saw the number of nodes in the TFG growing linearly with respect to the number of lines of code, not quadratically as indicated by the worst-case bounds.

In FLAVERS, the number of node-tuples in an analysis gives a good indication of the amount of memory needed to perform an analysis. In most of our examples, the number of node-tuples grew quadratically with the size of the system, indicating FLAVERS scales well with regards to space, in addition to time.

We attribute this performance to FLAVERS' approach of incrementally improving the model until it is precise enough to prove the property. It is well known that the cost of unreduced state-space enumeration techniques scales exponentially with the number of tasks [Tay83b]. Based on the performance of other finite-state verification approaches on these or similar programs, it is not the case that we are choosing "easy" programs as examples. FLAVERS is demonstrating polynomial growth on programs with state spaces that grow exponentially in the number of potential program executions. Unfortunately, we also saw situations where the rate of growth in FLAVERS' analysis time was unacceptable. Recent work has been unable to predict when FLAVERS or other finite-state verification techniques will have such undesirable performance [ACD+99].

Although FLAVERS' performance on the considered problems is often good, these programs are small and mostly drawn from the standard example programs used to test finite-state verification tools. The largest program analyzed was the Chiron example, which is about 15,000 lines long. Chiron is not a contrived program designed to test finite-state verification tools. Since FLAVERS' performance on Chiron was similar to its performance on the other programs, we have reason to believe that we will see comparable performance on other real world programs.

As mentioned previously, although our experiments give the cost of running FLAVERS on a system where a minimal set of constraints is known, they do not give any indication of the overall or human cost of applying FLAVERS to verify a property. In particular, we do not look at the cost of finding a set of constraints sufficient for proving that a property holds. In our experience, it is not too difficult to find such a set of constraints; it usually requires one to three iterations. Moreover, the cost of running FLAVERS for inconclusive problems is usually much smaller than running it on conclusive problems since our actual implementation stops as soon as one property violation is found. Thus, the cost for inconclusive problems is very variable and somewhat serendipitous. There are heuristics that appear to be very good at predicting the constraints that should be used [TAC04] and we are currently investigating heuristics that reduce the cost of finding an initial inconclusive result.

8 Related Work

As both a flow-analysis and a finite-state verification technique, FLAVERS draws on a long line of previous work. We focus our discussion here on the connections between FLAVERS and flow analysis of concurrent systems, model checking techniques, and systematic approaches to abstraction.

8.1 Flow Analysis

Program flow analysis was developed to enable optimizing compilers to generate efficient code [ASU85]. In flow analysis, specific patterns of behavior are analyzed by computing the fix-point of a relation defined over a graph structure whose nodes explicitly encode control information.

Most flow analyses are targeted at analyzing the system for a single property, such as the set of live variables at each system statement [ASU85]. FLAVERS uses an approach that combines multiple flow analyses into a single analysis. Combining multiple dependent flow analyses into a single flow analysis (e.g., [CC95, HR81]) offers the potential to improve the precision of analysis results, for example by eliminating some infeasible paths. In FLAVERS, analyses based on constraint automata are combined on-the-fly in such a way that any automaton reaching a violation state causes the entire combined analysis to reach a violation state. The strength of this approach is that it makes it easy to define analyses that are sensitive to selected semantic features of the program, while efficiently abstracting the balance of the program.

The bulk of flow-analysis research has been aimed at sequential software, but flow analysis of concurrent software is becoming increasingly important. Analyses that are both flow-insensitive and context-insensitive can be adapted directly to reason about concurrent software since, by definition, they do not reason about the ordering of statements either within or between tasks. Flow-sensitive analyses typically use flow-graph models that are, essentially, a collection of task flow graphs with additional edges, or labels, to represent inter-task synchronization and communication. These include, for example, analyses to detect the potential for statements to execute concurrently [MR93, NA98, NAC99b], data races [NM90], and pointer and escape analyses [SR01]. For more general analyses, however, interleavings need to either be modeled in the graph or incorporated into the analysis algorithm. The flow-graph model used in FLAVERS,

described in Section 4, explicitly represents the potential interleaving of pairs of events.

The application of flow-analysis techniques for validation of sequential code [FO76, Joh78, Ryd74] followed quite quickly after the development of flow analysis for optimization. These approaches used a set of analyses formulated to detect a fixed set of anomalies. The Cecil/Cesar system [OO90, OO92] generalized this approach by providing a single parameterized flow analysis for properties expressed as QREs. FLAVERS builds on this work by extending the analysis to concurrent systems and by incorporating a variety of mechanisms to increase the precision of the results.

More recently, the SLAM [BR01] and BLAST [HJMS02] projects have each adapted an existing context-sensitive flow-analysis algorithm [RHS95] to reason about correctness properties of sequential C programs that are specified as automata. Like FLAVERS, these projects have a flexible approach to incorporating selected semantic features of the program into their analysis. Unlike FLAVERS, both the SLAM and BLAST provide automated support for determining how the program should be abstracted. This work is similar to determining what constraints should be used in the FLAVERS, a direction of future work for us.

Flow analysis for validation of concurrent programs has been less well studied. Masticola and Ryder developed a system for checking deadlock freedom in Ada tasking systems [MR91, Mas93]. They incorporate an approach for refining the system model that is similar in spirit to the refinements described in Section 4. In their approach, they use flow analyses to refine the system model prior to the final analysis, thereby increasing the precision of the analysis results [MR93]. FLAVERS differs from this work in that it applies refinements only a single time, provides for more precise analysis through the use of constraints, and supports analysis of a wide class of properties as opposed to just deadlock.

8.2 Model Checking

Model checking refers to a class of techniques for checking conformance of the behaviors encoded in a finite-state transition system to a formula written in a temporal logic [MP91]. Model checking has gained popularity due to its success in reasoning about hardware components and communication protocols [CGP99].

The relationship between model checking and flow analysis is very strong; both involve a fix-point computation of a relation that encodes a specified property. In fact it has been shown that a wide range of useful flow analyses can be formulated as model-checking problems and vice versa [MOSS99, Sch98]. The application of model checking techniques to software relies on a pre-processing step for extracting a safe, compact model of a system's behavior. Only very recent approaches automate this step [BR01, CDH+00, DIS99, VHBP00]. Once a model is produced, properties of the model can be checked. In contrast, flow analyses are designed to automatically extract a compact model and then to analyze this model. FLAVERS applies both of these approaches by extracting a safe, compact model through the application of refinements, optionally extracting additional information through the use of constraints, and then performing the property analysis.

Explicit-state model checking involves the construction of the reachable state space of a transition system and the calculation of the formulas that are true at each state [CGP99]. Much of the early work on static concurrency analysis took this approach, although each limited their search to specific system states, such as states that violate assertions or describe deadlocks [Tay83b, YTL+95]. Unfortunately, the size of a system's reachable state space grows exponentially with the number of tasks, even when all data are ignored [Tay83a]. One popular form of explicit-state model checking is the automata-based approach employed in the SPIN model checker [Hol97]. In this approach the property to be checked is combined with the state space to reduce model checking to state reachability; for liveness properties the check involves a form of cycle detection. FLAVERS can be viewed as performing a kind of explicit-state analysis. In fact, if constraints are used for each system task and system variable then FLAVERS will construct the full reachable state space encoded in the sets of tuples propagated to the TFG nodes. The strength of FLAVERS is that it provides a very flexible mechanism for defining abstracted state spaces that are very compact and for which analysis is relatively inexpensive.

To increase the applicability of explicit-state analysis, researchers have investigated a variety of techniques including partial-order reductions that detect equivalent states and then limit analysis to a single representative of the equivalence class (e.g., [Pel98]) and compositional approaches that build and analyze the state space compositionally, (e.g., [CPS93, YY91, CK96]). Although none of these techniques have been able to reduce the worst-case complexity of analysis to be sub-exponential, each technique has been successfully applied to selected systems. FLAVERS in-

corporates a form of partial-order reduction [NCC99] and has been used to perform assume-guarantee compositional reasoning [Dwy97].

Information about the execution environment of the system under analysis can help reduce the number of reachable system states. Cheung and Kramer model behaviors of execution environments with *context constraints* in the context of explicit compositional model checking [CK96]. In their approach, context constraints are FSAs that are composed with the FSAs modeling components of the system under analysis. These context constraints are quite similar to FLAVERS' constraints from Section 6 and in fact our use of constraints to model the execution environment is identical to their approach. FLAVERS' constraints are more general in encoding different semantics aspects of a program. For example, VA constraints are derived from an abstract finite semantics of a variable's data type.

There is a rich literature on non-enumerative state space analyses. Some of the most well-studied are the symbolic model checking approaches [McM93]. These operate by manipulating an encoding of the next-state function using some form of decision diagram, typically an ordered binary decision diagram [Bry92]. Fix-point calculations are performed using this representation to determine the set of states that satisfy a given formula. For certain systems, symbolic encoding can yield reductions in the space and time required for model checking by several orders of magnitude. Unfortunately, it has proven very difficult to predict whether such reductions will be effective for a given system and property to be analyzed. Other non-enumerative techniques can offer similar advantages, for example deductive methods based on theorem proving, (e.g., [HJvdB99, FLL+02]) and methods based on integer programming [Cor96].

8.3 State-space Abstraction

Fundamental decidability results force all precise concurrency analyses to employ reductions or abstractions to achieve tractability. Flow analyses encode abstractions that preserve only the property under analysis. Flow analyses expressed as abstract interpretations [CC77] make the nature of the abstraction explicit. Recent work has applied abstract interpretation to the generation of abstracted state spaces that are amenable to model checking.

In predicate abstraction, one provides predicates over a transition system's state variables and a new transition system is derived that has a boolean state variable for each predicate [GS97]. Scaling this approach to programming languages is a challenge, and several partial solutions have been investigated (e.g., [BR01, DHJ+01]). FLAVERS can accommodate these kinds of abstractions through the use of constraints. For instance, an integer variable could be modeled by a constraint automaton with five states: *unknown*, *positive*, *negative*, *zero*, and *violation*. Transitions in the automaton would be defined appropriately for operations on the variable, such as tests of and assignments to the variable.

9 Conclusion

We have presented FLAVERS, a finite-state verification approach that analyzes whether software systems satisfy user-defined properties. In contrast to other finite-state verification techniques, FLAVERS creates a system model that is relatively small but still conservative with respect to the property that is being evaluated. FLAVERS achieves this reduction in size at the cost of precision. Analysts can improve the precision of the results as needed by selectively and judiciously incorporating additional semantic information into the analysis problem. FLAVERS provides automated support for creating some of the common constraints that are used to represent this additional information.

Our evaluation, although preliminary, indicates that sufficient precision can usually be achieved relatively easily and that the cost for many problems grows as a low-order polynomial in the size of the system. In addition to the evaluation reported here, FLAVERS has been applied in a number of interesting ways. It has been used in an empirical evaluation of concurrency analysis techniques [Cha96] and to analyze communication protocols [NCO96], partial software systems [Dwy97], and architectural system descriptions [NACO97]. It has also been applied to advanced distributed simulation systems [BMD96] and to evaluate the adherence of such systems to high-level architectural requirements [Cor97]. In these latter two studies, FLAVERS detected previously unknown errors in the systems being analyzed. As is often the case with rigorous verification techniques, many of the errors were found just because of the additional scrutiny associated with formulating the properties and annotating the code. Although more errors were detected while preparing the associated artifacts than while using verification, the errors that FLAVERS did detect

were deemed to be more complicated then those caught during preparation. In other words, the errors that FLAVERS detected could not be easily observed by the analyst without such validation tools.

Although this paper has described the approach in some detail for Ada systems, the approach is relatively language independent. Each programming language must be carefully translated into the system model so that the ordering of events is conservatively captured in the TFG's flow of control. To date, translators have been successfully developed by others for C++ and Jovial. We are currently exploring some alternative models for Java [NAC99a]. Interestingly, changes to the state-propagation algorithm to support these different languages were relatively minor. The MHP algorithm, however, seems to require some careful specialization for programming languages with different models of concurrency [NA98, NAC99b].

In this paper, we have emphasized how FLAVERS can be used to verify properties of software systems. In addition to being applicable to programs, the FLAVERS approach is applicable to other artifacts that capture the flow of events through a system. For example, it could be applied to architectural descriptions or detailed designs. When applied to programs, the verification process could be viewed as a complementary activity to testing. Alternatively, it could be used to help with debugging, where a hypothesis about a fault is formulated as a property and FLAVERS is then used to determine if there are any traces through the TFG (and thus consequently through the code) that would cause this fault to occur. Such support is particularly important for distributed systems, since output results may vary depending on the run-time task scheduling.

There are some limitations to the approach used in FLAVERS. The model is less precise than reachability-graph based approaches [Hol97, CK93] and thus does not seem well suited for the kinds of detailed analysis required for deadlock detection and definition/reference anomaly detection. Moreover, it is restricted to properties that can be represented in a regular language. Being a static analysis approach, it also cannot reason about dynamic configurations. Thus, as was done in our experimental evaluation, specific configurations must be selected. Users can often present compelling arguments that verification over a restricted set of configurations is equivalent to verification over the general set, but this is not always possible or easy. Others have argued, however, that most faults are detected with small sample sizes [JV00].

The current implementation of FLAVERS/Ada does not support dynamically-allocated data, recursion, exceptions, generics, or some of the low-level representation data types. There has been considerable work on modeling recursion for data-flow analysis [RHS95], but using these techniques to model multi-threaded systems is not practical [Ram00]. By reducing the flow-graph representation based on the property to be checked, FLAVERS may localize the need to model recursion to a single thread, and we are studying the extent to which existing techniques can be applied in this case. The other limitations primarily pose engineering hurdles that could be addressed with appropriate effort.

There are several interesting research directions to be explored. Improving performance is always of concern. We are particularly interested in exploring techniques for decomposing the analysis problem. Because the events of interest tend to be sparsely located throughout the system, the resulting model tends to be small after alphabet refinement and other optimizations are applied. In all the programs that we have examined, using inlining to create the model worked reasonably well since the optimized representation for an invoked component was usually very small, and often was empty. This will not always be the case, however, so we are exploring how counterexamples can be used to automate assume-guarantee reasoning using the L* learning algorithm [CDP03].

Counterexample generation also raises some interesting issues. We have studied how different variations of the state-propagation algorithm can impact performance, depending on whether consistent or inconsistent results are expected or if short counterexample traces are desired [CCO01]. Path length is only one of the important attributes of counterexamples, however. Another concern is determining if the counterexample is executable. One interesting approach, used in SLAM [BR01], is to employ symbolic evaluation techniques to help determine path executability. We are currently exploring techniques for guiding the selection of counterexamples using the constraint mechanism, not only for path feasibility but also as a debugging aid.

Specifying properties is also an area of future work. It is surprisingly difficult to capture correctly the intent of a system. Although regular expressions and FSA models seem more natural to practitioners than temporal logics, many subtle options need to be considered when formulating a property. A recent experimental study showed that many finite-state verification properties could be mapped to a small number of property patterns [DAC99]. Building on that work, a framework for extending these patterns into more extensively parameterized templates has been developed. This framework provides FSA-based templates as well as a natural-language interface for describing prop-

erties [SACO02]. The hope is that this framework will help practitioners formulate properties in a way that is more natural and intuitive for them, but at the same time will provide a rigorous mathematical representation that can be used as a basis for verification.

Finally, model development is yet another area of future research that we have been actively exploring. One issue is whether the iterative model development approach used by FLAVERS can be supplanted with an approach that extensively exploits compiler optimizations, partial evaluation, and abstraction techniques. The Bandera system is being developed to explore this issue [CDH+00]. It is employing a number of techniques, such as program slicing, to automatically create a concise model optimized for the property and system configuration. This direction is also being explored in the work on Java PathFinder [VHBP00]. The resulting model might be sufficiently precise that incremental improvement might not be necessary. Alternatively, this concise model might be a better starting point for the incremental refinement approach used by FLAVERS. We are currently developing a version of FLAVERS where the TFG is built from the highly optimized internal representation produced by Bandera. We are also exploring alternative representations, such as maintaining some of the variable information directly in the TFG representation as well as doing some dynamic expression evaluation during the analysis process itself.

Although there are a number of interesting research issues to be explored, the FLAVERS toolset demonstrates the feasibility of applying flow-analysis based verification techniques to software systems. It is extremely important that practical techniques be developed to help reason about software systems. Distributed systems are particularly problematic, and are becoming extremely common. Such systems are more difficult to reason about, to test, and to debug than sequential systems, for which these problems are difficult enough. Although the experimental evaluation reported here is preliminary and the example programs are relativley modest in size, the experimental results are promising and lead us to believe that finite-state verification techniques, in general, and the FLAVERS approach, in particular, hold considerable promise for helping developers reason about systems.

Acknowledgements

The authors would like to thank Michael Sutherland of the UMass Statistical Consulting Center for his help in analyzing the experimental results and Heather Conboy, Stephen Siegel, Rachel Smith, and Jianbin Tan for helping with several of the examples. Jay Corbett and Corina Păsăreanu contributed to the Chiron experiments. Leon Osterweil and George Avrunin have been valued colleagues, providing insightful comments over the years.

References

[ABC+91] G.S. Avrunin, U.A. Buy, J.C. Corbett, L.K. Dillon, and J.C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, November 1991.

[ACD+99] G.S. Avrunin, J.C. Corbett, M.B. Dwyer, C.S. Păsăreanu, and S.F. Siegel. Comparing finite-state verification techniques for concurrent software. TR 99-69, University of Massachusetts, Department of Computer Science, November 1999.

[ASU85] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[BMD96] C. Bouwens, R. McKenzie, and C. Dean. Investigating static data flow analysis for advanced distributed simulation verification. In *Proceedings of the 15th Workshop in the Interoperability of Distributed Interactive Simulation*, pp. 473–478, September 1996.

[BR01] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Model Checking Software : Proceedings of the 8th International SPIN Workshop*, May 2001.

[Bry92] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[CA94] J.C. Corbett and G.S. Avrunin. Towards scalable compositional analysis. *Software Engineering Notes*, 19(5):53–61, December 1994. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pp. 238–252, 1977.

[CC95] C. Click and K.D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

[CCO01] J.M. Cobleigh, L.A. Clarke, and L.J. Osterweil. The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. In *Proceedings of the* 23rd *International Conference on Software Engineering*, pp. 37–46, May 2001.

[CDH+00] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[CDP03] J.M. Cobleigh, D.Giannakopoulou, and C.S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 331–346, April 2003.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.

[Cha96] A.T. Chamillard. *An Empirical Comparison of Static Concurrency Analysis Techniques*. PhD thesis, University of Massachusetts at Amherst, May 1996.

[CK93] S.C. Cheung and J. Kramer. Tractable flow analysis for anomaly detection in distributed programs. In *Proceedings of the European Software Engineering Conference*, 1993.

[CK96] S.C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Programming Languages and Systems*, 5(7):334–377, October 1996.

[Cor96] J.C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), March 1996.

[Cor97] Science Applications International Corporation. Advanced interoperability technology development: Investigating static data flow analysis for advanced distributed simulation verification. Technical report, SAIC, Orlando, FL, May 1997.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[DAC99] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of* 21st *International Conference on Software Engineering*, May 1999.

[DBDS93] S. Duri, U. Buy, R. Devarapalli, and S.M. Shatz. Using state space methods for deadlock analysis in Ada tasking. *Software Engineering Notes*, 18(3):51–60, July 1993. Proceedings of the International Symposium on Software Testing and Analysis.

[DHJ+01] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[DIS99] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software–Practice and Experience*, 29(7):577–603, June 1999.

[Dwy95] M.B. Dwyer. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs*. PhD thesis, University of Massachusetts at Amherst, September 1995.

[Dwy97] M.B. Dwyer. Modular flow analysis for concurrent software. In *Proceeding of the 12th IEEE Conference on Automated Software Engineering*, November 1997.

[FLL+02] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.

[FO76] L.D. Fosdick and L.J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proceedings 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Sci-*

ence, pp. 72–83, June 1997.

[Hec77] M.S. Hecht. *Flow Analysis of Computer Programs*. The Computer Science Library Programming Language Series. Elsevier North-Holland, 1977.

[HJMS02] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pp. 58–70, 2002.

[HJvdB99] M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. In *Object-Oriented Technology: ECOOP'99 Workshop Reader*, volume 1743 of *Lecture Notes in Computer Science*, pp. 109–110. Springer-Verlag, June 1999.

[HL85] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[Hoa69] C.A.R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[Hol00] G.J. Holzmann. Logic verification of ANSI-C code with SPIN. In *The Seventh International SPIN Workshop on Model Checking of Software (SPIN'2000)*, pp. 131–147, September 2000.

[How86] W.E. Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, SE-12:997–1004, October 1986.

[HR81] L.H. Holley and B.K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, SE-7(1):60–78, January 1981.

[Joh78] S.C. Johnson. Lint, a C program checker. In *Unix Programmer's Manual, AT&T Bell Laboratories*, 1978.

[JV00] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proceedings of the 2000 International Symposium on Software Testing and Analysis*, August 2000.

[KCTT91] 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*, pp. 208–218, May 1991.

[Mas93] S.P. Masticola. *Static Detection of Deadlocks in Polynomial Time*. PhD thesis, Rutgers University, May 1993.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.

[MMR95] S.P. Masticola, T.J. Marlowe, and B.G. Ryder. Lattice frameworks for multisource and bidirectional data flow problems. *ACM Transactions on Programming Languages and Systems*, 17(5):777–803, September 1995.

[MOSS99] M. Müller-Olm, D.A. Schmidt, and B. Steffen. Model checking: a tutorial introduction. In *Proceedings of the 6th Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pp. 330–354, 1999.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[MR90] T.J. Marlowe and B.G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[MR91] S.P. Masticola and B.G. Ryder. A model of Ada programs for static deadlock detection in polynomial time. In *Proceedings of Workshop on Parallel and Distributed Debugging*. ACM, May 1991.

[MR93] S.P. Masticola and B.G. Ryder. Non-concurrency analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[NA98] G.N. Naumovich and G.S. Avrunin. A conservative data flow algorithm for detecting all pairs of stat ements that may happen in parallel. In *Proceedings of the Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, November 1998.

[NAC99a] G.N. Naumovich, G.S. Avrunin, and L.A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of 21st International Conference on Software Engineering*, May 1999.

[NAC99b] G.N. Naumovich, G.S. Avrunin, and L.A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the ESEC/FSE 99, Joint 7th European Software*

Engineering Conference(ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7), September 1999.

[NACO97] G.N. Naumovich, G.S. Avrunin, L.A. Clarke, and L.J. Osterweil. Applying static analysis to software architectures. In *LNCS 1301*. The 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 1997.

[NC00] G.N. Naumovich and L.A. Clarke. Classifying properties: An alternative to the safety-liveness classification. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-00)*, volume 25, 6 of *ACM Software Engineering Notes*, pp. 159–168, November 2000.

[NCC99] G.N. Naumovich, L.A. Clarke, and J. Cobleigh. Using partial order techniques to improve performance of data flow analysis based verification. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, volume 24, 5 of *Software Engeneering Notes (SEN)*, pp. 57–65, September 1999.

[NCO96] G.N. Naumovich, L.A. Clarke, and L.J. Osterweil. Verification of communication protocols using data flow analysis. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1996.

[NCO98] G.N. Naumovich, L.A. Clarke, and L.J. Osterweil. Efficient composite data flow analysis applied to concurrent programs. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, June 1998.

[NM90] R.H. Netzer and B.P. Miller. Detecting data races in parallel program executions. In *Proceedings of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[OO90] K.M. Olender and L.J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16(3):268–280, March 1990.

[OO92] K.M. Olender and L.J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, January 1992.

[Pel98] D.A. Peled. Ten years of partial order reduction. In *CAV, Computer Aided Verification*, number 1427 in LNCS, pp. 17–28, Vancouver, BC, Canada, 1998. Springer.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, March 2000.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 49–61, 1995.

[Ryd74] B.G. Ryder. The PFORT verifier. *Software: Practice and Experience*, 4(4):359–377, 1974.

[SACO02] R.L. Smith, G.S. Avrunin, L.A. Clarke, and L.J. Osterweil. Propel: An approach supporting property elucidation. In *Proceedings of the 24rd International Conference on Software Engineering*, pp. 11–21, May 2002.

[Sch98] D.A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 38–48, 1998.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[SR01] A. Salcianu and M.C. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2001.

[TAC04] J. Tan, G.S. Avrunin, and L.A. Clarke. Heuristic-based model refinement for FLAVERS. In *Proceedings of the 26th International Conference on Software Engineering*, May 2004.

[Tay83a] R.N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[Tay83b] R.N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26, May 1983.

[TBC+88] R.N. Taylor, F.C. Belz, L.A. Clarke, L.J. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young. Foundations for the Arcadia Environment Architecture. In *Proceedings of SIGSOFT88: Third Symposium*

on Software Development Environment, pp. 1–13, November 1988. Published as ACM SIGPLAN Notices 24(2) and as SIGSOFT Software Engineering Notes, 13(5) November 1988.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S.J. Park. Model checking programs. In *Proceedings of the Fifteenth ASE IEEE International Conference on Automated Software Engineering*, September 2000.

[YTL+95] M. Young, R.N. Taylor, D.L. Levine, K.A. Nies, and D. Brodbeck. A concurrency analysis tool suite: Rationale, design, and preliminary experience. *ACM Transactions on Software Engineering and Methodology*, 4(1):64–106, January 1995.

[YY91] W.J. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis and Verification*, pp. 49–59, Victoria, Canada, October 1991.

A Proofs

This appendix contains proofs of conservativeness of the reduced CFG model, TFG model, and the state-propagation algorithm of FLAVERS and the proof of complexity of the state propagation algorithm.

A.1 RCFG Conservativeness

The following lemma and theorem prove that the CFG alphabet refinement is conservative. We call nodes in the CFG that are removed by the refinement algorithm, and thus do not appear in the RCFG, *removed* and all other nodes *retained*. In the lemma, we refer to two CFGs, G and G'. We use superscripts to differentiate between different elements of the tuples of G and G'. For example, N^G refers to the set of nodes of G and $N^{G'}$ refers to the set of nodes of G'. We use this convention to disambiguate the origin of an element when necessary.

Lemma 8. *Let G be a CFG and G' the RCFG obtained from G using the alphabet refinement algorithm. For any pair of retained nodes m, n from N^G with corresponding nodes m' and n' in $N^{G'}$, if there is a path from m to n in G that consists only of removed nodes, then there is an edge (m', n') in $E^{G'}$.*

Proof. The proof is by induction on the length of the path. If this path is of length 1, i.e. $(m, n) \in E^G$, then trivially $(m', n') \in E^{G'}$, since only those edges that are incident to a removed node are removed by the alphabet-refinement algorithm.

Assume that the statement of the theorem holds for paths of length k. Consider any path of length $k + 1$: $m, n_1, \ldots, n_k, n$. Let $n_i \in \{n_1, \ldots, n_k\}$ be the first of the nodes $n_1, \ldots, n_k$ removed by the alphabet-refinement algorithm. Then, since n_{i-1} is a predecessor of n_i and n_{i+1} is a successor of n, an edge (n_{i-1}, n_{i+1}) will be created (temporarily, if $n_{i-1} \neq m$ or $n_{i+1} \neq n$). After this, the length of the path between m and n is of length k and, according to the induction hypothesis, eventually an edge between m and n is created. □

Theorem 1 (RCFG Conservativeness). *Let G be a CFG and G' the corresponding RCFG obtained using the alphabet-refinement algorithm. Let $R \subseteq N^G$ be the set of nodes of G that were retained in G'. For any sequence of nodes $\pi \in (N^G)^*$, define Map$(\pi) \in (N^{G'})^*$ as the sequence of nodes in G' that corresponds to the nodes in the projection $\pi|_R$. We need to show that $\pi \in G$ implies Map$(\pi) \in G'$.*

Proof. We argue by induction on the number of nodes in π. The statement of the theorem trivially holds if π is a path of length 0. Suppose the statement of the theorem holds for any path of length k. Consider any path of length $k + 1$. It can be written as πn where π is a path of length k and $n \in N^G$ is the final node in the path.

First, suppose $n \notin R$. Then $\pi|_R = \pi n|_R$. By our inductive hypothesis, we know that *Map*$(\pi) \in G'$, so *Map*$(\pi n) \in G'$, as desired.

Second, suppose $n \in R$. There are two cases to consider. Let $\pi|_R$ be non-empty. Then let $m \in N^G$ be the last node in the path corresponding to π such that $m \in R$. Nodes m and n are connected by a path of removed nodes. By Lemma 8, there is an edge $(m', n') \in E^{G'}$. Using the inductive hypothesis, *Map*$(\pi|_R) \in G'$. Since $(m', n') \in E^{G'}$, *Map*$(\pi n) \in G'$, as desired.

Now let $\pi|_R$ be empty. According to the CFG alphabet-refinement algorithm in Figure 6, the initial node $n_{initial}$ of G is retained. All nodes between $n_{initial}$ and n are removed and so, again by Lemma 8, $(n'_{initial}, n') \in E^{G'}$ and $Map(\pi n) \in G'$, as desired. □

A.2 TFG Conservativeness

The following theorem states and proves conservativeness of the TFG model:

Theorem 2 (TFG Conservativeness). *For each actual execution of the system S, the TFG contains a path starting in the initial node that exhibits the same set of events projected on Σ_I as this execution projected on Σ_I. Thus, $\mathcal{L}(S) \subseteq \mathcal{L}(G)$.*

Proof. For easy correlation between the system and the TFG, we use TFG nodes to represent a unit of system execution. This could be viewed as introducing a unique event to mark each node in the TFG and tracking these events in the system. Thus, in this proof we say "a node is executed" to mean that the code corresponding to this node is executed. The system executes a communication node when the synchronization between the two tasks represented by this communication node takes place. The proof is by induction on the length of the execution.

Executions of length 1 correspond to execution of a single node, which must be the initial node $n^i_{initial}$ in one of the system tasks T^i. Since the initial node of the TFG is connected to each of the initial nodes of the tasks, the corresponding path in the TFG is $n_{initial}, n^i_{initial}$.

As an inductive hypothesis, suppose that the statement of the theorem holds for executions of length k. Consider any execution of length $k+1$: $n_{initial}, n_1, \ldots, n_k, n_{k+1}$. By the induction hypothesis, there is a path $n_{initial}, n_1, \ldots, n_k$ in the TFG. We consider several cases based on the relationship between n_k and n_{k+1}.

- **$\mathbf{n_k}$ and $\mathbf{n_{k+1}}$ are in the same task $\mathbf{T^i}$.** There are several possibilities to consider.
  - First, assume that both n_k and n_{k+1} are local nodes, but neither corresponds to a task synchronization. Since they happen subsequently on an actual execution of the system, the code corresponding to n_k precedes the code corresponding to n_{k+1} in the control flow of T^i. Since, by Theorem 1, RCFGs are conservative representations of individual tasks, edge (n_k, n_{k+1}) is in the RCFG for T^i. By the TFG construction, this edge is also in the TFG, and so path $n_{initial}, n_1, \ldots, n_k, n_{k+1}$ is in the TFG.
  - Second, suppose n_k is a local node that corresponds to a task synchronization statement of task T^i. The next event in the system in task T^i after a synchronization statement is the actual synchronization with another task. Therefore, n_{k+1} must be a communication node. By the TFG construction, this is indeed the case, and $(n_k, n_{k+1}) \in E_{com}$. Therefore, path $n_{initial}, n_1, \ldots, n_k, n_{k+1}$ is in the TFG.
  - Third, suppose that n_k corresponds to a task synchronization, which means that $n_k \in N_{com}$. Then n_{k+1} must be a local node in task T^i that immediately follows the synchronization represented by n_k. By the TFG construction, $(n_k, n_{k+1}) \in E_{com}$. Therefore, path $n_{initial}, n_1, \ldots, n_k, n_{k+1}$ is in the TFG.
  - Finally, by the TFG construction, n_k and n_{k+1} cannot both be communication nodes.
- **$\mathbf{n_k}$ and $\mathbf{n_{k+1}}$ belong to different sets of tasks.** Let n_k belong to task T^i and n_{k+1} belong to task T^j where $i \neq j$. This case represents a context switch between two tasks, where executions of T^i and T^j are interleaved. The MHP algorithm is conservative [NA98] and therefore it detects that n_k and n_{k+1} may happen in parallel. Since construction of MIP edges is directly based on the MHP algorithm, $(n_k, n_{k+1}), (n_{k+1}, n_k) \in E_{mip}$. Therefore, path $n_{initial}, n_1, \ldots, n_k, n_{k+1}$ is in the TFG.
- **$\mathbf{n_{k+1}}$ is the final node $\mathbf{n_{final}}$.** In this case, the execution $n_{initial}, n_1, \ldots, n_k, n_{k+1}$ represents a terminal execution of the system, which means that the task RCFGs execute their final nodes, and so n_k is one of these final nodes n^i_{final}. Since the TFG contains an edge from each n^i_{final} to the final node of the TFG n_{final}, the path corresponding to this execution is in the TFG.

□

Note that the reverse of the statement of this theorem is not true. There may exist paths in the TFG that do not correspond to actual executions of the system. For example, in Figure 8, partial path $n_{initial}, 1, 21, 2$ does not correspond

to an actual execution of the system, because the synchronization represented at node 21 cannot happen until node 10 in task ATM executes. We address this source of imprecision with a special type of constraint, described in Section 6.

A.3 State-propagation Conservativeness

The following theorem proves conservativeness of the state-propagation algorithm of FLAVERS.

Theorem 5 (Conservativeness of State Propagation). *For any TFG node n, if there exists an actual program execution on which state r of the property is associated with n, then after the state-propagation algorithm terminates, $r \in$ States$[n]$.*

Proof. According to Theorem 2 about conservativeness of the TFG representation, there is a path $\pi = n_0, \ldots, n_k$, where $n_0 = n_{initial}$, that corresponds to the actual execution described in the statement of this theorem. Let s_i be the state of the property on node n_i of the path. We use an argument by induction on the length of π.

Suppose first that the length of π is 0, meaning it consists of only $n_{initial}$. Since the initial node represents the state of the program before the execution begins, the only state of the property that can correspond to this state of the program is the start state, s. According to the initialization step of the state-propagation algorithm, $s \in$ States$[n_{initial}]$.

Assume that the statement of the theorem holds for executions that correspond to paths of length k. Consider the case where the length of π is $k+1$. By the induction hypothesis, state s_k that is associated with n_k on path π is in States$[n_k]$. Because States$[n_k]$ must change at some point during the algorithm, n_{k+1} is placed on the worklist at least once. When the state-propagation algorithm processes node n_{k+1}, all states associated with each of n_{k+1}'s predecessors are propagated through n_{k+1}. Therefore, state s_{k+1} will be placed in States$[n_{k+1}]$. □

Next, we show that the results of state propagation with constraints is conservative. This will be proved in two parts. The first part shows that state propagation with constraints where pruning of tuples is not performed when a constraint automaton enters its violation state is conservative. Next, we show that the pruning of tuples does not change the conservativeness of the results of state propagation. For the first part of the proof, we need to change the definition of $F^n(x)$, the transfer function for a TFG node n and a set of tuples X, from the one given in Subsection 6.3 to one that does not prune tuples:

$$\forall n \in N, \forall X \subseteq \textit{Tuples}, F^n(X) = \{f^n(T) \mid T \in X\} \tag{1}$$

Corollary 9 (Conservativeness of State Propagation with Constraints and without Pruning Tuples). *Given a TFG, a property automaton P, and a set of constraint automata $C_1, C_2, \ldots, C_k$, for any TFG node n, if there exists an actual program execution on which a tuple $T = (t^P, t^{C_1}, t^{C_2}, \ldots, t^{C_k})$ is associated with n, then after the state-propagation algorithm terminates using the propagation function given in Equation 1, T is associated with n.*

Proof. The proof of this corollary is similar to the proof of Theorem 5. □

Theorem 10 (Conservativeness of State Propagation with Constraints and with Pruning Tuples). *Given a TFG, a property automaton P, and a set of constraint automata $C_1, C_2, \ldots, C_k$, for any TFG node n, if there exists an actual program execution on which a tuple $T = (t^P, t^{C_1}, t^{C_2}, \ldots, t^{C_k})$, such that $\forall i, 1 \leq i \leq k, t^{C_i} \neq v^{C_i}$, is associated with n_{final}, then after the state-propagation algorithm terminates using the propagation functions from Section 6.3, T is associated with n_{final}.*

Proof. The proof of this theorem follows from that of Corollary 9 if we can prove that when the transfer function without pruning (Equation 1) is used, then if a tuple $T_1 = ({t_1}^P, {t_1}^{C_1}, {t_1}^{C_2}, \ldots, {t_1}^{C_k})$ that has one of its constraint automata C_i in the violation state, i.e. for some $1 \leq i \leq k$, ${t_1}^{C_i} = v^{C_i}$, is associated with node n of the TFG, then the propagation of tuple T_1 does not result in a tuple $T_2 = ({t_2}^P, {t_2}^{C_1}, {t_2}^{C_2}, \ldots, {t_2}^{C_k})$ being associated with n_{final}, such that ${t_2}^{C_i} \neq v^{C_i}$. If such a tuple T_2 could be associated with n_{final}, then state propagation would examine T_2 when determining if the property holds and removing T_1 would be unsafe. There are two cases to consider:

- First, suppose that $n \neq n_{final}$. Then if tuple T_1 is propagated along a path $n, n_1, \ldots, n_k, n_{final}$, then the tuple associated with n_{final} would be $T_2 = f^{n_{final}}(f^{n_k}(\cdots f^{n_1}(T_1)\cdots))$. If ${t_1}^{C_i} = v^{C_i}$, then, according to the

propagation function in Equation 1 and because all transitions out of a violation state are self-transitions, ${t_2}^{C_i} = v^{C_i}$.

- Second, suppose that $n = n_{final}$. Then, since the final node has no outgoing edges, there is no way that T_1 can be propagated to any other nodes.

Since pruning tuples with constraints in violation states does not affect the result of state propagation, state propagation with constraints and with pruning tuples is conservative.

□

A.4 Complexity of the State-Propagation Algorithm

The following theorem proves the complexity result for the state propagation algorithm from Figure 10.

Theorem 6 (State-propagation Complexity). *Given a TFG, with nodes N^G, and a property automaton, with states S^P, state-propagation algorithm terminates in $\mathcal{O}(|S^P||N^G|^2)$ time.*

Proof. We prove that the complexity of recomputing sets *States*, *IN*, V, U, and *Flow* for any node n, amortized over the course of the algorithm, is $\mathcal{O}(|N^G||S^P|)$. Since there are $|N^G|$ nodes in the TFG, the statements of the theorem will follow.

The total number of worklist insertions is $\mathcal{O}(|E^G||S^P|)$, because a node is added to the worklist only if the *States* set of one of its predecessors changes and each such set can change at most $|S^P|$ times. Since $|E^G| \leq |N^G|^2$, the complexity of maintaining the worklist falls within $\mathcal{O}(|S^P||N^G|^2)$.

Consider any node $n \in N^G$. In a run of the algorithm, *IN*$[n]$ has at most $|S^P|$ states inserted from each of its, at most, $|N^G| - 1$ predecessors. There are a constant number of set operations (such as checking membership in a set and insertion and deletion from a set) performed at each step. Each of these set operations takes a constant time. Therefore, over the course of the algorithm, there are $\mathcal{O}(|N^G||S^P|)$ set operations performed for each node. Since there are $|N^G|$ nodes in the TFG, the total number of set operations is $\mathcal{O}(|S^P||N^G|^2)$.

Determining the results is still bounded by the number of states in the property, $|S^P|$. The total complexity of this algorithm is $\mathcal{O}(|S^P||N^G|^2 + |S^P|) = \mathcal{O}(|S^P||N^G|^2)$. □

B Size Information

This section gives size information for all of the property checks presented in the body of the paper. Data is presented in terms of several measures of program and property check size. LOC gives the number of lines of code for the Ada program, not counting blank lines and lines that contain only comments; Nodes and Edges are the number of nodes and edges of the TFG; Tuples is the number of different tuples created during state propagation; and Node-Tuples is the sum, for each node in the TFG, of the number of tuples on that node.

B.1 The Reader/Writer Program

Readers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	75	26	51	16	46
2	97	40	89	24	78
3	119	54	127	32	110
4	141	68	165	40	142
5	163	82	203	48	174
6	185	96	241	56	206
7	207	110	279	64	238

Table 13: Size Information for **exclusive read write**

Readers	LOC	Nodes	Edges	Tuples	Node-Tuples
8	229	124	317	72	270
9	251	138	355	80	302
10	273	152	393	88	334
11	295	166	431	96	366
12	317	180	469	104	398
13	339	194	507	112	430
14	361	208	545	120	462
15	383	222	583	128	494
20	493	292	773	168	654
25	603	362	963	208	814
30	713	432	1,153	248	974
35	823	502	1,343	288	1,134
40	933	572	1,533	328	1,294
45	1,043	642	1,723	368	1,454
50	1,153	712	1,913	408	1,614
55	1,263	782	2,103	448	1,774
60	1,373	852	2,293	488	1,934
65	1,483	922	2,483	528	2,094
70	1,593	992	2,673	568	2,254
75	1,703	1,062	2,863	608	2,414
80	1,813	1,132	3,053	648	2,574
85	1,923	1,202	3,243	688	2,734
90	2,033	1,272	3,433	728	2,894
95	2,143	1,342	3,623	768	3,054
100	2,253	1,412	3,813	808	3,214
105	2,363	1,482	4,003	848	3,374
110	2,473	1,552	4,193	888	3,534
115	2,583	1,622	4,383	928	3,694
120	2,693	1,692	4,573	968	3,854
125	2,803	1,762	4,763	1,008	4,014
130	2,913	1,832	4,953	1,048	4,174
135	3,023	1,902	5,143	1,088	4,334
140	3,133	1,972	5,333	1,128	4,494
145	3,243	2,042	5,523	1,168	4,654
150	3,353	2,112	5,713	1,208	4,814
155	3,463	2,182	5,903	1,248	4,974
160	3,573	2,252	6,093	1,288	5,134
165	3,683	2,322	6,283	1,328	5,294
170	3,793	2,392	6,473	1,368	5,454
175	3,903	2,462	6,663	1,408	5,614
180	4,013	2,532	6,853	1,448	5,774
185	4,123	2,602	7,043	1,488	5,934
190	4,233	2,672	7,233	1,528	6,094
195	4,343	2,742	7,423	1,568	6,254
200	4,453	2,812	7,613	1,608	6,414
n	$22n + 53$	$14n + 12$	$38n + 13$	$8n + 8$	$32n + 14$

Table 13: Size Information for **exclusive read write**

Readers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	75	25	50	16	47
2	97	39	88	33	103
3	119	53	126	56	177
4	141	67	164	85	269
5	163	81	202	120	379
6	185	95	240	161	507
7	207	109	278	208	653
8	229	123	316	261	817
9	251	137	354	320	999
10	273	151	392	385	1,199
15	383	221	582	800	2,469
20	493	291	772	1,365	4,189
25	603	361	962	2,080	6,359
30	713	431	1,152	2,945	8,979
35	823	501	1,342	3,960	12,049
40	933	571	1,532	5,125	15,569
45	1,043	641	1,722	6,440	19,539
50	1,153	711	1,912	7,905	23,959
55	1,263	781	2,102	9,520	28,829
60	1,373	851	2,292	11,285	34,149
65	1,483	921	2,482	13,200	39,919
70	1,593	991	2,672	15,265	46,139
75	1,703	1,061	2,862	17,480	52,809
80	1,813	1,131	3,052	19,845	59,929
85	1,923	1,201	3,242	22,360	67,499
90	2,033	1,271	3,432	25,025	75,519
95	2,143	1,341	3,622	27,840	83,989
100	2,253	1,411	3,812	30,805	92,909
105	2,363	1,481	4,002	33,920	102,279
110	2,473	1,551	4,192	37,185	112,099
115	2,583	1,621	4,382	40,600	122,369
120	2,693	1,691	4,572	44,165	133,089
125	2,803	1,761	4,762	47,880	144,259
130	2,913	1,831	4,952	51,745	155,879
135	3,023	1,901	5,142	55,760	167,949
140	3,133	1,971	5,332	59,925	180,469
145	3,243	2,041	5,522	64,240	193,439
150	3,353	2,111	5,712	68,705	206,859
155	3,463	2,181	5,902	73,320	220,729
160	3,573	2,251	6,092	78,085	235,049
165	3,683	2,321	6,282	83,000	249,819
170	3,793	2,391	6,472	88,065	265,039
175	3,903	2,461	6,662	93,280	280,709
180	4,013	2,531	6,852	98,645	296,829
185	4,123	2,601	7,042	104,160	313,399
190	4,233	2,671	7,232	109,825	330,419
195	4,343	2,741	7,422	115,640	347,889

Table 14: Size Information for **no write while reading**

Readers	LOC	Nodes	Edges	Tuples	Node-Tuples
200	4,453	2,811	7,612	121,605	365,809
n	$22n + 53$	$14n + 11$	$38n + 12$	$3n^2 + 8n + 5$	$9n^2 + 29n + 9$

Table 14: Size Information for **no write while reading**

Readers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	75	25	50	16	47
2	97	39	88	25	81
3	119	53	126	34	115
4	141	67	164	43	149
5	163	81	202	52	183
6	185	95	240	61	217
7	207	109	278	70	251
8	229	123	316	79	285
9	251	137	354	88	319
10	273	151	392	97	353
11	295	165	430	106	387
12	317	179	468	115	421
13	339	193	506	124	455
14	361	207	544	133	489
15	383	221	582	142	523
20	493	291	772	187	693
25	603	361	962	232	863
30	713	431	1,152	277	1,033
35	823	501	1,342	322	1,203
40	933	571	1,532	367	1,373
45	1,043	641	1,722	412	1,543
50	1,153	711	1,912	457	1,713
55	1,263	781	2,102	502	1,883
60	1,373	851	2,292	547	2,053
65	1,483	921	2,482	592	2,223
70	1,593	991	2,672	637	2,393
75	1,703	1,061	2,862	682	2,563
80	1,813	1,131	3,052	727	2,733
85	1,923	1,201	3,242	772	2,903
90	2,033	1,271	3,432	817	3,073
95	2,143	1,341	3,622	862	3,243
100	2,253	1,411	3,812	907	3,413
105	2,363	1,481	4,002	952	3,583
110	2,473	1,551	4,192	997	3,753
115	2,583	1,621	4,382	1,042	3,923
120	2,693	1,691	4,572	1,087	4,093
125	2,803	1,761	4,762	1,132	4,263
130	2,913	1,831	4,952	1,177	4,433
135	3,023	1,901	5,142	1,222	4,603
140	3,133	1,971	5,332	1,267	4,773

Table 15: Size Information for **no write while reading 2**

Readers	LOC	Nodes	Edges	Tuples	Node-Tuples
145	3,243	2,041	5,522	1,312	4,943
150	3,353	2,111	5,712	1,357	5,113
155	3,463	2,181	5,902	1,402	5,283
160	3,573	2,251	6,092	1,447	5,453
165	3,683	2,321	6,282	1,492	5,623
170	3,793	2,391	6,472	1,537	5,793
175	3,903	2,461	6,662	1,582	5,963
180	4,013	2,531	6,852	1,627	6,133
185	4,123	2,601	7,042	1,672	6,303
190	4,233	2,671	7,232	1,717	6,473
195	4,343	2,741	7,422	1,762	6,643
200	4,453	2,811	7,612	1,807	6,813
n	$22n + 53$	$14n + 11$	$38n + 12$	$9n + 7$	$34n + 13$

Table 15: Size Information for **no write while reading 2**

Readers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	75	21	45	9	38
2	97	35	85	15	70
3	119	49	125	21	102
4	141	63	165	27	134
5	163	77	205	33	166
6	185	91	245	39	198
7	207	105	285	45	230
8	229	119	325	51	262
9	251	133	365	57	294
10	273	147	405	63	326
11	295	161	445	69	358
12	317	175	485	75	390
13	339	189	525	81	422
14	361	203	565	87	454
15	383	217	605	93	486
20	493	287	805	123	646
25	603	357	1,005	153	806
30	713	427	1,205	183	966
35	823	497	1,405	213	1,126
40	933	567	1,605	243	1,286
45	1,043	637	1,805	273	1,446
50	1,153	707	2,005	303	1,606
55	1,263	777	2,205	333	1,766
60	1,373	847	2,405	363	1,926
65	1,483	917	2,605	393	2,086
70	1,593	987	2,805	423	2,246
75	1,703	1,057	3,005	453	2,406
80	1,813	1,127	3,205	483	2,566
85	1,923	1,197	3,405	513	2,726

Table 16: Size Information for **write first**

Readers	LOC	Nodes	Edges	Tuples	Node-Tuples
90	2,033	1,267	3,605	543	2,886
95	2,143	1,337	3,805	573	3,046
100	2,253	1,407	4,005	603	3,206
105	2,363	1,477	4,205	633	3,366
110	2,473	1,547	4,405	663	3,526
115	2,583	1,617	4,605	693	3,686
120	2,693	1,687	4,805	723	3,846
125	2,803	1,757	5,005	753	4,006
130	2,913	1,827	5,205	783	4,166
135	3,023	1,897	5,405	813	4,326
140	3,133	1,967	5,605	843	4,486
145	3,243	2,037	5,805	873	4,646
150	3,353	2,107	6,005	903	4,806
155	3,463	2,177	6,205	933	4,966
160	3,573	2,247	6,405	963	5,126
165	3,683	2,317	6,605	993	5,286
170	3,793	2,387	6,805	1,023	5,446
175	3,903	2,457	7,005	1,053	5,606
180	4,013	2,527	7,205	1,083	5,766
185	4,123	2,597	7,405	1,113	5,926
190	4,233	2,667	7,605	1,143	6,086
195	4,343	2,737	7,805	1,173	6,246
200	4,453	2,807	8,005	1,203	6,406
n	$22n + 53$	$14n + 7$	$40n + 5$	$6n + 3$	$32n + 6$

Table 16: Size Information for **write first**

B.2 The Gas Station Program

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
2	138	55	144	179	751
3	187	76	258	1,258	6,347
4	236	97	396	8,289	47,661
5	285	118	558	51,908	330,493
6	334	139	744	313,087	2,170,599
7	383	160	954	1,836,450	13,714,835

Table 17: Size Information for **mutual exclusion**, 1 pump

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
2	227	94	448	901	13,675
3	311	132	976	14,169	346,331
4	395	170	1,680	201,637	6,814,799
5	479	208	2,560	2,720,705	117,309,035

Table 18: Size Information for **mutual exclusion**, 2 pumps

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
2	316	133	880	2,101	52,361
3	435	188	2,078	51,165	2,162,142
4	554	243	3,732	1,126,693	67,393,343

Table 19: Size Information for **mutual exclusion**, 3 pumps

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	89	33	53	2	33
2	138	54	125	2	54
3	187	75	221	2	75
4	236	96	341	2	96
5	285	117	485	2	117
6	334	138	653	2	138
7	383	159	845	2	159
8	432	180	1,061	2	180
9	481	201	1,301	2	201
10	530	222	1,565	2	222
15	775	327	3,245	2	327
20	1,020	432	5,525	2	432
25	1,265	537	8,405	2	537
30	1,510	642	11,885	2	642
35	1,755	747	15,965	2	747
40	2,000	852	20,645	2	852
45	2,245	957	25,925	2	957
50	2,490	1,062	31,805	2	1,062
55	2,735	1,167	38,285	2	1,167
60	2,980	1,272	45,365	2	1,272
65	3,225	1,377	53,045	2	1,377
70	3,470	1,482	61,325	2	1,482
75	3,715	1,587	70,205	2	1,587
80	3,960	1,692	79,685	2	1,692
85	4,205	1,797	89,765	2	1,797
90	4,450	1,902	100,445	2	1,902
95	4,695	2,007	111,725	2	2,007
100	4,940	2,112	123,605	2	2,112
105	5,185	2,217	136,085	2	2,217
110	5,430	2,322	149,165	2	2,322
115	5,675	2,427	162,845	2	2,427
120	5,920	2,532	177,125	2	2,532
125	6,165	2,637	192,005	2	2,637
130	6,410	2,742	207,485	2	2,742
135	6,655	2,847	223,565	2	2,847
140	6,900	2,952	240,245	2	2,952
145	7,145	3,057	257,525	2	3,057
150	7,390	3,162	275,405	2	3,162
155	7,635	3,267	293,885	2	3,267
160	7,880	3,372	312,965	2	3,372

Table 20: Size Information for **customer 1 start/stop**, 1 pump

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
165	8,125	3,477	332,645	2	3,477
170	8,370	3,582	352,925	2	3,582
175	8,615	3,687	373,805	2	3,687
180	8,860	3,792	395,285	2	3,792
185	9,105	3,897	417,365	2	3,897
190	9,350	4,002	440,045	2	4,002
195	9,595	4,107	463,325	2	4,107
200	9,840	4,212	487,205	2	4,212
n	$49n + 40$	$21n + 12$	$12n^2 + 36n + 5$	2	$21n + 12$

Table 20: Size Information for **customer 1 start/stop**, 1 pump

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	143	55	95	2	55
2	227	93	375	2	93
3	311	131	831	2	131
4	395	169	1,463	2	169
5	479	207	2,271	2	207
6	563	245	3,255	2	245
7	647	283	4,415	2	283
8	731	321	5,751	2	321
9	815	359	7,263	2	359
10	899	397	8,951	2	397
15	1,319	587	20,031	2	587
20	1,739	777	35,511	2	777
25	2,159	967	55,391	2	967
30	2,579	1,157	79,671	2	1,157
35	2,999	1,347	108,351	2	1,347
40	3,419	1,537	141,431	2	1,537
45	3,839	1,727	178,911	2	1,727
50	4,259	1,917	220,791	2	1,917
55	4,679	2,107	267,071	2	2,107
60	5,099	2,297	317,751	2	2,297
65	5,519	2,487	372,831	2	2,487
70	5,939	2,677	432,311	2	2,677
75	6,359	2,867	496,191	2	2,867
80	6,779	3,057	564,471	2	3,057
85	7,199	3,247	637,151	2	3,247
90	7,619	3,437	714,231	2	3,437
95	8,039	3,627	795,711	2	3,627
100	8,459	3,817	881,591	2	3,817
105	8,879	4,007	971,871	2	4,007
110	9,299	4,197	1,066,551	2	4,197
115	9,719	4,387	1,165,631	2	4,387
120	10,139	4,577	1,269,111	2	4,577
125	10,559	4,767	1,376,991	2	4,767

Table 21: Size Information for **customer 1 start/stop**, 2 pumps

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
130	10,979	4,957	1,489,271	2	4,957
135	11,399	5,147	1,605,951	2	5,147
140	11,819	5,337	1,727,031	2	5,337
145	12,239	5,527	1,852,511	2	5,527
150	12,659	5,717	1,982,391	2	5,717
155	13,079	5,907	2,116,671	2	5,907
160	13,499	6,097	2,255,351	2	6,097
165	13,919	6,287	2,398,431	2	6,287
170	14,339	6,477	2,545,911	2	6,477
175	14,759	6,667	2,697,791	2	6,667
180	15,179	6,857	2,854,071	2	6,857
185	15,599	7,047	3,014,751	2	7,047
190	16,019	7,237	3,179,831	2	7,237
195	16,439	7,427	3,349,311	2	7,427
200	16,859	7,617	3,523,191	2	7,617
n	$84n + 59$	$38n + 17$	$88n^2 + 16n - 9$	2	$38n + 17$

Table 21: Size Information for **customer 1 start/stop**, 2 pumps

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	197	77	137	2	77
2	316	132	753	2	132
3	435	187	1,825	2	187
4	554	242	3,353	2	242
5	673	297	5,337	2	297
6	792	352	7,777	2	352
7	911	407	10,673	2	407
8	1,030	462	14,025	2	462
9	1,149	517	17,833	2	517
10	1,268	572	22,097	2	572
15	1,863	847	50,257	2	847
20	2,458	1,122	89,817	2	1,122
25	3,053	1,397	140,777	2	1,397
30	3,648	1,672	203,137	2	1,672
35	4,243	1,947	276,897	2	1,947
40	4,838	2,222	362,057	2	2,222
45	5,433	2,497	458,617	2	2,497
50	6,028	2,772	566,577	2	2,772
55	6,623	3,047	685,937	2	3,047
60	7,218	3,322	816,697	2	3,322
65	7,813	3,597	958,857	2	3,597
70	8,408	3,872	1,112,417	2	3,872
75	9,003	4,147	1,277,377	2	4,147
80	9,598	4,422	1,453,737	2	4,422
85	10,193	4,697	1,641,497	2	4,697
90	10,788	4,972	1,840,657	2	4,972

Table 22: Size Information for **customer 1 start/stop**, 3 pumps

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
95	11,383	5,247	2,051,217	2	5,247
100	11,978	5,522	2,273,177	2	5,522
105	12,573	5,797	2,506,537	2	5,797
110	13,168	6,072	2,751,297	2	6,072
115	13,763	6,347	3,007,457	2	6,347
120	14,358	6,622	3,275,017	2	6,622
125	14,953	6,897	3,553,977	2	6,897
130	15,548	7,172	3,844,337	2	7,172
135	16,143	7,447	4,146,097	2	7,447
140	16,738	7,722	4,459,257	2	7,722
145	17,333	7,997	4,783,817	2	7,997
150	17,928	8,272	5,119,777	2	8,272
155	18,523	8,547	5,467,137	2	8,547
160	19,118	8,822	5,825,897	2	8,822
165	19,713	9,097	6,196,057	2	9,097
170	20,308	9,372	6,577,617	2	9,372
175	20,903	9,647	6,970,577	2	9,647
180	21,498	9,922	7,374,937	2	9,922
185	22,093	10,197	7,790,697	2	10,197
190	22,688	10,472	8,217,857	2	10,472
195	23,283	10,747	8,656,417	2	10,747
200	23,878	11,022	9,106,377	2	11,022
n	$119n + 78$	$55n + 22$	$228n^2 - 68n - 23$	2	$55n + 22$

Table 22: Size Information for **customer 1 start/stop**, 3 pumps

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	89	32	52	10	72
2	138	53	142	17	282
3	187	74	262	24	549
4	236	95	418	31	904
5	285	116	610	38	1,347
6	334	137	838	45	1,878
7	383	158	1,102	52	2,497
8	432	179	1,402	59	3,204
9	481	200	1,738	66	3,999
10	530	221	2,110	73	4,882
15	775	326	4,510	108	10,617
20	1,020	431	7,810	143	18,552
25	1,265	536	12,010	178	28,687
30	1,510	641	17,110	213	41,022
35	1,755	746	23,110	248	55,557
40	2,000	851	30,010	283	72,292
45	2,245	956	37,810	318	91,227
50	2,490	1,061	46,510	353	112,362
55	2,735	1,166	56,110	388	135,697

Table 23: Size Information for **pump 1 start/stop**, 1 pump

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
60	2,980	1,271	66,610	423	161,232
65	3,225	1,376	78,010	458	188,967
70	3,470	1,481	90,310	493	218,902
75	3,715	1,586	103,510	528	251,037
80	3,960	1,691	117,610	563	285,372
85	4,205	1,796	132,610	598	321,907
90	4,450	1,901	148,510	633	360,642
95	4,695	2,006	165,310	668	401,577
100	4,940	2,111	183,010	703	444,712
105	5,185	2,216	201,610	738	490,047
110	5,430	2,321	221,110	773	537,582
115	5,675	2,426	241,510	808	587,317
120	5,920	2,531	262,810	843	639,252
125	6,165	2,636	285,010	878	693,387
130	6,410	2,741	308,110	913	749,722
135	6,655	2,846	332,110	948	808,257
140	6,900	2,951	357,010	983	868,992
145	7,145	3,056	382,810	1,018	931,927
150	7,390	3,161	409,510	1,053	997,062
155	7,635	3,266	437,110	1,088	1,064,397
160	7,880	3,371	465,610	1,123	1,133,932
165	8,125	3,476	495,010	1,158	1,205,667
170	8,370	3,581	525,310	1,193	1,279,602
175	8,615	3,686	556,510	1,228	1,355,737
180	8,860	3,791	588,610	1,263	1,434,072
185	9,105	3,896	621,610	1,298	1,514,607
190	9,350	4,001	655,510	1,333	1,597,342
195	9,595	4,106	690,310	1,368	1,682,277
200	9,840	4,211	726,010	1,403	1,769,412
$n > 1$	$49n + 40$	$21n + 11$	$18n^2 + 30n + 10$	$7n + 3$	$44n^2 + 47n + 12$

Table 23: Size Information for **pump 1 start/stop**, 1 pump

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	143	54	94	10	186
2	227	92	446	17	971
3	311	130	998	24	1,992
4	395	168	1,774	31	3,377
5	479	206	2,774	38	5,126
6	563	244	3,998	45	7,239
7	647	282	5,446	52	9,716
8	731	320	7,118	59	12,557
9	815	358	9,014	66	15,762
10	899	396	11,134	73	19,331
15	1,319	586	25,094	108	42,636
20	1,739	776	44,654	143	75,041

Table 24: Size Information for **pump 1 start/stop**, 2 pumps

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
25	2,159	966	69,814	178	116,546
30	2,579	1,156	100,574	213	167,151
35	2,999	1,346	136,934	248	226,856
40	3,419	1,536	178,894	283	295,661
45	3,839	1,726	226,454	318	373,566
50	4,259	1,916	279,614	353	460,571
55	4,679	2,106	338,374	388	556,676
60	5,099	2,296	402,734	423	661,881
65	5,519	2,486	472,694	458	776,186
70	5,939	2,676	548,254	493	899,591
75	6,359	2,866	629,414	528	1,032,096
80	6,779	3,056	716,174	563	1,173,701
85	7,199	3,246	808,534	598	1,324,406
90	7,619	3,436	906,494	633	1,484,211
95	8,039	3,626	1,010,054	668	1,653,116
100	8,459	3,816	1,119,214	703	1,831,121
105	8,879	4,006	1,233,974	738	2,018,226
110	9,299	4,196	1,354,334	773	2,214,431
115	9,719	4,386	1,480,294	808	2,419,736
120	10,139	4,576	1,611,854	843	2,634,141
125	10,559	4,766	1,749,014	878	2,857,646
130	10,979	4,956	1,891,774	913	3,090,251
135	11,399	5,146	2,040,134	948	3,331,956
140	11,819	5,336	2,194,094	983	3,582,761
145	12,239	5,526	2,353,654	1,018	3,842,666
150	12,659	5,716	2,518,814	1,053	4,111,671
155	13,079	5,906	2,689,574	1,088	4,389,776
160	13,499	6,096	2,865,934	1,123	4,676,981
165	13,919	6,286	3,047,894	1,158	4,973,286
170	14,339	6,476	3,235,454	1,193	5,278,691
175	14,759	6,666	3,428,614	1,228	5,593,196
180	15,179	6,856	3,627,374	1,263	5,916,801
185	15,599	7,046	3,831,734	1,298	6,249,506
190	16,019	7,236	4,041,694	1,333	6,591,311
195	16,439	7,426	4,257,254	1,368	6,942,216
200	16,859	7,616	4,478,414	1,403	7,302,221
$n > 1$	$84n + 59$	$38n + 16$	$112n^2 - 8n + 14$	$7n + 3$	$182n^2 + 111n + 21$

Table 24: Size Information for **pump 1 start/stop**, 2 pumps

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	197	76	136	10	292
2	316	131	878	17	1,580
3	435	186	2,118	24	3,258
4	554	241	3,898	31	5,538
5	673	296	6,218	38	8,420

Table 25: Size Information for **pump 1 start/stop**, 3 pumps

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
6	792	351	9,078	45	11,904
7	911	406	12,478	52	15,990
8	1,030	461	16,418	59	20,678
9	1,149	516	20,898	66	25,968
10	1,268	571	25,918	73	31,860
15	1,863	846	59,118	108	70,350
20	2,458	1,121	105,818	143	123,890
25	3,053	1,396	166,018	178	192,480
30	3,648	1,671	239,718	213	276,120
35	4,243	1,946	326,918	248	374,810
40	4,838	2,221	427,618	283	488,550
45	5,433	2,496	541,818	318	617,340
50	6,028	2,771	669,518	353	761,180
55	6,623	3,046	810,718	388	920,070
60	7,218	3,321	965,418	423	1,094,010
65	7,813	3,596	1,133,618	458	1,283,000
70	8,408	3,871	1,315,318	493	1,487,040
75	9,003	4,146	1,510,518	528	1,706,130
80	9,598	4,421	1,719,218	563	1,940,270
85	10,193	4,696	1,941,418	598	2,189,460
90	10,788	4,971	2,177,118	633	2,453,700
95	11,383	5,246	2,426,318	668	2,732,990
100	11,978	5,521	2,689,018	703	3,027,330
105	12,573	5,796	2,965,218	738	3,336,720
110	13,168	6,071	3,254,918	773	3,661,160
115	13,763	6,346	3,558,118	808	4,000,650
120	14,358	6,621	3,874,818	843	4,355,190
125	14,953	6,896	4,205,018	878	4,724,780
130	15,548	7,171	4,548,718	913	5,109,420
135	16,143	7,446	4,905,918	948	5,509,110
140	16,738	7,721	5,276,618	983	5,923,850
145	17,333	7,996	5,660,818	1,018	6,353,640
150	17,928	8,271	6,058,518	1,053	6,798,480
155	18,523	8,546	6,469,718	1,088	7,258,370
160	19,118	8,821	6,894,418	1,123	7,733,310
165	19,713	9,096	7,332,618	1,158	8,223,300
170	20,308	9,371	7,784,318	1,193	8,728,340
175	20,903	9,646	8,249,518	1,228	9,248,430
180	21,498	9,921	8,728,218	1,263	9,783,570
185	22,093	10,196	9,220,418	1,298	10,333,760
190	22,688	10,471	9,726,118	1,333	10,899,000
195	23,283	10,746	10,245,318	1,368	11,479,290
200	23,878	11,021	10,778,018	1,403	12,074,630
$n > 1$	$119n + 78$	$55n + 21$	$270n^2 - 110n + 18$	$7n + 3$	$301n^2 + 173n + 30$

Table 25: Size Information for **pump 1 start/stop**, 3 pumps

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	143	54	98	98	278
2	227	92	370	1,022	7,076
3	311	130	818	2,970	41,164
4	395	168	1,442	5,858	108,449
5	479	206	2,242	9,718	224,576
6	563	244	3,218	14,550	402,721
7	647	282	4,370	20,354	656,060
8	731	320	5,698	27,130	997,769
9	815	358	7,202	34,878	1,441,024
10	899	396	8,882	43,598	1,999,001
15	1,319	586	19,922	101,778	6,970,876
20	1,739	776	35,362	184,258	16,787,201
25	2,159	966	55,202	291,038	33,094,976
30	2,579	1,156	79,442	422,118	57,541,201
35	2,999	1,346	108,082	577,498	91,772,876
40	3,419	1,536	141,122	757,178	137,437,001
45	3,839	1,726	178,562	961,158	196,180,576
46	3,923	1,764	186,578	1,004,870	209,643,761
$n > 2$	$84n + 59$	$38n + 16$	$88n^2 + 8n + 2$	$486n^2 - 514n + 138$	$2196n^3 - 1931n^2 - 450n + 601$

Table 26: Size Information for **correct change**, 2 pumps

Customers	LOC	Nodes	Edges	Tuples	Node-Tuples
1	197	76	140	187	567
2	316	131	738	2,638	30,946
3	435	186	1,792	7,647	255,257
4	554	241	3,302	15,122	672,864
5	673	296	5,268	25,117	1,395,485
6	792	351	7,690	37,632	2,506,280
7	911	406	10,568	52,667	4,088,409
8	1,030	461	13,902	70,222	6,225,032
9	1,149	516	17,692	90,297	8,999,309
10	1,268	571	21,938	112,892	12,494,400
15	1,863	846	50,008	263,667	43,692,665
20	2,458	1,121	89,478	477,442	105,385,280
25	3,053	1,396	140,348	754,217	207,967,245
$n > 2$	$119n + 78$	$55n + 21$	$228n^2 - 86n - 2$	$1260n^2 - 1345n + 342$	$13860n^3 - 13813n^2 + 1478n + 920$

Table 27: Size Information for **correct change**, 3 pumps

B.3 The Dining Philosophers Program

B.3.1 Standard

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
3	79	47	143	39	695

Table 28: Size Information for **mutual exclusion**

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
4	104	61	280	39	1,153
5	129	75	449	39	1,611
6	154	89	650	39	2,069
7	179	103	883	39	2,527
8	204	117	1,148	39	2,985
9	229	131	1,445	39	3,443
10	254	145	1,774	39	3,901
15	379	215	3,899	39	6,191
20	504	285	6,824	39	8,481
25	629	355	10,549	39	10,771
30	754	425	15,074	39	13,061
35	879	495	20,399	39	15,351
40	1,004	565	26,524	39	17,641
45	1,129	635	33,449	39	19,931
50	1,254	705	41,174	39	22,221
55	1,379	775	49,699	39	24,511
60	1,504	845	59,024	39	26,801
65	1,629	915	69,149	39	29,091
70	1,754	985	80,074	39	31,381
75	1,879	1,055	91,799	39	33,671
80	2,004	1,125	104,324	39	35,961
85	2,129	1,195	117,649	39	38,251
90	2,254	1,265	131,774	39	40,541
95	2,379	1,335	146,699	39	42,831
100	2,504	1,405	162,424	39	45,121
105	2,629	1,475	178,949	39	47,411
110	2,754	1,545	196,274	39	49,701
115	2,879	1,615	214,399	39	51,991
120	3,004	1,685	233,324	39	54,281
125	3,129	1,755	253,049	39	56,571
130	3,254	1,825	273,574	39	58,861
135	3,379	1,895	294,899	39	61,151
140	3,504	1,965	317,024	39	63,441
145	3,629	2,035	339,949	39	65,731
150	3,754	2,105	363,674	39	68,021
155	3,879	2,175	388,199	39	70,311
160	4,004	2,245	413,524	39	72,601
165	4,129	2,315	439,649	39	74,891
170	4,254	2,385	466,574	39	77,181
175	4,379	2,455	494,299	39	79,471
180	4,504	2,525	522,824	39	81,761
185	4,629	2,595	552,149	39	84,051
190	4,754	2,665	582,274	39	86,341
195	4,879	2,735	613,199	39	88,631
200	5,004	2,805	644,924	39	90,921
n	$25n + 4$	$14n + 5$	$16n^2 + 25n - 76$	39	$458n - 679$

Table 28: Size Information for **mutual exclusion**

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
3	79	45	125	6	150
4	104	59	230	6	212
5	129	73	367	6	274
6	154	87	536	6	336
7	179	101	737	6	398
8	204	115	970	6	460
9	229	129	1,235	6	522
10	254	143	1,532	6	584
15	379	213	3,497	6	894
20	504	283	6,262	6	1,204
25	629	353	9,827	6	1,514
30	754	423	14,192	6	1,824
35	879	493	19,357	6	2,134
40	1,004	563	25,322	6	2,444
45	1,129	633	32,087	6	2,754
50	1,254	703	39,652	6	3,064
55	1,379	773	48,017	6	3,374
60	1,504	843	57,182	6	3,684
65	1,629	913	67,147	6	3,994
70	1,754	983	77,912	6	4,304
75	1,879	1,053	89,477	6	4,614
80	2,004	1,123	101,842	6	4,924
85	2,129	1,193	115,007	6	5,234
90	2,254	1,263	128,972	6	5,544
95	2,379	1,333	143,737	6	5,854
100	2,504	1,403	159,302	6	6,164
105	2,629	1,473	175,667	6	6,474
110	2,754	1,543	192,832	6	6,784
115	2,879	1,613	210,797	6	7,094
120	3,004	1,683	229,562	6	7,404
125	3,129	1,753	249,127	6	7,714
130	3,254	1,823	269,492	6	8,024
135	3,379	1,893	290,657	6	8,334
140	3,504	1,963	312,622	6	8,644
145	3,629	2,033	335,387	6	8,954
150	3,754	2,103	358,952	6	9,264
155	3,879	2,173	383,317	6	9,574
160	4,004	2,243	408,482	6	9,884
165	4,129	2,313	434,447	6	10,194
170	4,254	2,383	461,212	6	10,504
175	4,379	2,453	488,777	6	10,814
180	4,504	2,523	517,142	6	11,124
185	4,629	2,593	546,307	6	11,434
190	4,754	2,663	576,272	6	11,744
195	4,879	2,733	607,037	6	12,054
200	5,004	2,803	638,602	6	12,364
n	$25n + 4$	$14n + 3$	$16n^2 - 7n + 2$	6	$62n - 36$

Table 29: Size Information for **no fork 1 up twice**

B.3.2 Fork Manager

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
2	82	28	103	71	123
3	112	34	166	103	262
4	142	40	197	135	393
5	172	46	228	167	524
6	202	52	259	199	655
7	232	58	290	231	786
8	262	64	321	263	917
9	292	70	352	295	1,048
10	322	76	383	327	1,179
15	472	106	538	487	1,834
20	622	136	693	647	2,489
25	772	166	848	807	3,144
30	922	196	1,003	967	3,799
35	1,072	226	1,158	1,127	4,454
40	1,222	256	1,313	1,287	5,109
45	1,372	286	1,468	1,447	5,764
50	1,522	316	1,623	1,607	6,419
55	1,672	346	1,778	1,767	7,074
60	1,822	376	1,933	1,927	7,729
65	1,972	406	2,088	2,087	8,384
70	2,122	436	2,243	2,247	9,039
75	2,272	466	2,398	2,407	9,694
80	2,422	496	2,553	2,567	10,349
85	2,572	526	2,708	2,727	11,004
90	2,722	556	2,863	2,887	11,659
95	2,872	586	3,018	3,047	12,314
100	3,022	616	3,173	3,207	12,969
105	3,172	646	3,328	3,367	13,624
110	3,322	676	3,483	3,527	14,279
115	3,472	706	3,638	3,687	14,934
120	3,622	736	3,793	3,847	15,589
125	3,772	766	3,948	4,007	16,244
130	3,922	796	4,103	4,167	16,899
135	4,072	826	4,258	4,327	17,554
140	4,222	856	4,413	4,487	18,209
145	4,372	886	4,568	4,647	18,864
150	4,522	916	4,723	4,807	19,519
155	4,672	946	4,878	4,967	20,174
160	4,822	976	5,033	5,127	20,829
165	4,972	1,006	5,188	5,287	21,484
170	5,122	1,036	5,343	5,447	22,139
175	5,272	1,066	5,498	5,607	22,794
180	5,422	1,096	5,653	5,767	23,449
185	5,572	1,126	5,808	5,927	24,104
190	5,722	1,156	5,963	6,087	24,759

Table 30: Size Information for **mutual exclusion**

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
195	5,872	1,186	6,118	6,247	25,414
200	6,022	1,216	6,273	6,407	26,069
$n > 2$	$30n + 22$	$6n + 16$	$31n + 73$	$32n + 7$	$131n - 131$

Table 30: Size Information for **mutual exclusion**

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
2	82	30	49	20	39
3	112	36	64	24	56
4	142	42	79	28	73
5	172	48	94	32	90
6	202	54	109	36	107
7	232	60	124	40	124
8	262	66	139	44	141
9	292	72	154	48	158
10	322	78	169	52	175
15	472	108	244	72	260
20	622	138	319	92	345
25	772	168	394	112	430
30	922	198	469	132	515
35	1,072	228	544	152	600
40	1,222	258	619	172	685
45	1,372	288	694	192	770
50	1,522	318	769	212	855
55	1,672	348	844	232	940
60	1,822	378	919	252	1,025
65	1,972	408	994	272	1,110
70	2,122	438	1,069	292	1,195
75	2,272	468	1,144	312	1,280
80	2,422	498	1,219	332	1,365
85	2,572	528	1,294	352	1,450
90	2,722	558	1,369	372	1,535
95	2,872	588	1,444	392	1,620
100	3,022	618	1,519	412	1,705
105	3,172	648	1,594	432	1,790
110	3,322	678	1,669	452	1,875
115	3,472	708	1,744	472	1,960
120	3,622	738	1,819	492	2,045
125	3,772	768	1,894	512	2,130
130	3,922	798	1,969	532	2,215
135	4,072	828	2,044	552	2,300
140	4,222	858	2,119	572	2,385
145	4,372	888	2,194	592	2,470
150	4,522	918	2,269	612	2,555
155	4,672	948	2,344	632	2,640
160	4,822	978	2,419	652	2,725
165	4,972	1,008	2,494	672	2,810

Table 31: Size Information for **no fork 1 up twice**

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
170	5,122	1,038	2,569	692	2,895
175	5,272	1,068	2,644	712	2,980
180	5,422	1,098	2,719	732	3,065
185	5,572	1,128	2,794	752	3,150
190	5,722	1,158	2,869	772	3,235
195	5,872	1,188	2,944	792	3,320
200	6,022	1,218	3,019	812	3,405
n	$30n + 22$	$6n + 18$	$15n + 19$	$4n + 12$	$17n + 5$

Table 31: Size Information for **no fork 1 up twice**

B.3.3 Dictionary

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
2	71	40	75	38	130
3	104	57	208	95	1,790
4	137	74	409	95	3,282
5	170	91	658	95	4,694
6	203	108	955	95	6,106
7	236	125	1,300	95	7,518
8	269	142	1,693	95	8,930
9	302	159	2,134	95	10,342
10	335	176	2,623	95	11,754
15	500	261	5,788	95	18,814
20	665	346	10,153	95	25,874
25	830	431	15,718	95	32,934
30	995	516	22,483	95	39,994
35	1,160	601	30,448	95	47,054
40	1,325	686	39,613	95	54,114
45	1,490	771	49,978	95	61,174
50	1,655	856	61,543	95	68,234
55	1,820	941	74,308	95	75,294
60	1,985	1,026	88,273	95	82,354
65	2,150	1,111	103,438	95	89,414
70	2,315	1,196	119,803	95	96,474
75	2,480	1,281	137,368	95	103,534
80	2,645	1,366	156,133	95	110,594
85	2,810	1,451	176,098	95	117,654
90	2,975	1,536	197,263	95	124,714
95	3,140	1,621	219,628	95	131,774
100	3,305	1,706	243,193	95	138,834
105	3,470	1,791	267,958	95	145,894
110	3,635	1,876	293,923	95	152,954
115	3,800	1,961	321,088	95	160,014
120	3,965	2,046	349,453	95	167,074
125	4,130	2,131	379,018	95	174,134
130	4,295	2,216	409,783	95	181,194

Table 32: Size Information for **mutual exclusion**

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
135	4,460	2,301	441,748	95	188,254
140	4,625	2,386	474,913	95	195,314
145	4,790	2,471	509,278	95	202,374
150	4,955	2,556	544,843	95	209,434
155	5,120	2,641	581,608	95	216,494
160	5,285	2,726	619,573	95	223,554
165	5,450	2,811	658,738	95	230,614
170	5,615	2,896	699,103	95	237,674
175	5,780	2,981	740,668	95	244,734
180	5,945	3,066	783,433	95	251,794
185	6,110	3,151	827,398	95	258,854
190	6,275	3,236	872,563	95	265,914
195	6,440	3,321	918,928	95	272,974
200	6,605	3,406	966,493	95	280,034
$n > 3$	$33n + 5$	$17n + 6$	$24n^2 + 33n - 107$	95	$1412n - 2366$

Table 32: Size Information for **mutual exclusion**

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
2	71	38	73	6	127
3	104	55	182	6	204
4	137	72	343	6	281
5	170	89	552	6	358
6	203	106	809	6	435
7	236	123	1,114	6	512
8	269	140	1,467	6	589
9	302	157	1,868	6	666
10	335	174	2,317	6	743
15	500	259	5,282	6	1,128
20	665	344	9,447	6	1,513
25	830	429	14,812	6	1,898
30	995	514	21,377	6	2,283
35	1,160	599	29,142	6	2,668
40	1,325	684	38,107	6	3,053
45	1,490	769	48,272	6	3,438
50	1,655	854	59,637	6	3,823
55	1,820	939	72,202	6	4,208
60	1,985	1,024	85,967	6	4,593
65	2,150	1,109	100,932	6	4,978
70	2,315	1,194	117,097	6	5,363
75	2,480	1,279	134,462	6	5,748
80	2,645	1,364	153,027	6	6,133
85	2,810	1,449	172,792	6	6,518
90	2,975	1,534	193,757	6	6,903
95	3,140	1,619	215,922	6	7,288
100	3,305	1,704	239,287	6	7,673
105	3,470	1,789	263,852	6	8,058

Table 33: Size Information for **no fork 1 up twice**

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
110	3,635	1,874	289,617	6	8,443
115	3,800	1,959	316,582	6	8,828
120	3,965	2,044	344,747	6	9,213
125	4,130	2,129	374,112	6	9,598
130	4,295	2,214	404,677	6	9,983
135	4,460	2,299	436,442	6	10,368
140	4,625	2,384	469,407	6	10,753
145	4,790	2,469	503,572	6	11,138
150	4,955	2,554	538,937	6	11,523
155	5,120	2,639	575,502	6	11,908
160	5,285	2,724	613,267	6	12,293
165	5,450	2,809	652,232	6	12,678
170	5,615	2,894	692,397	6	13,063
175	5,780	2,979	733,762	6	13,448
180	5,945	3,064	776,327	6	13,833
185	6,110	3,149	820,092	6	14,218
190	6,275	3,234	865,057	6	14,603
195	6,440	3,319	911,222	6	14,988
200	6,605	3,404	958,587	6	15,373
$n > 2$	$33n + 5$	$17n + 4$	$24n^2 - 7n - 13$	6	$77n - 27$

Table 33: Size Information for **no fork 1 up twice**

B.3.4 Host

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
2	87	44	132	87	445
3	114	62	305	87	2,118
4	141	80	564	87	3,496
5	168	98	887	87	4,874
6	195	116	1,274	87	6,252
7	222	134	1,725	87	7,630
8	249	152	2,240	87	9,008
9	276	170	2,819	87	10,386
10	303	188	3,462	87	11,764
15	438	278	7,637	87	18,654
20	573	368	13,412	87	25,544
25	708	458	20,787	87	32,434
30	843	548	29,762	87	39,324
35	978	638	40,337	87	46,214
40	1,113	728	52,512	87	53,104
45	1,248	818	66,287	87	59,994
50	1,383	908	81,662	87	66,884
55	1,518	998	98,637	87	73,774
60	1,653	1,088	117,212	87	80,664
65	1,788	1,178	137,387	87	87,554
70	1,923	1,268	159,162	87	94,444

Table 34: Size Information for **mutual exclusion**

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
75	2,058	1,358	182,537	87	101,334
80	2,193	1,448	207,512	87	108,224
85	2,328	1,538	234,087	87	115,114
90	2,463	1,628	262,262	87	122,004
95	2,598	1,718	292,037	87	128,894
100	2,733	1,808	323,412	87	135,784
105	2,868	1,898	356,387	87	142,674
110	3,003	1,988	390,962	87	149,564
115	3,138	2,078	427,137	87	156,454
120	3,273	2,168	464,912	87	163,344
125	3,408	2,258	504,287	87	170,234
130	3,543	2,348	545,262	87	177,124
135	3,678	2,438	587,837	87	184,014
140	3,813	2,528	632,012	87	190,904
145	3,948	2,618	677,787	87	197,794
150	4,083	2,708	725,162	87	204,684
155	4,218	2,798	774,137	87	211,574
160	4,353	2,888	824,712	87	218,464
165	4,488	2,978	876,887	87	225,354
170	4,623	3,068	930,662	87	232,244
175	4,758	3,158	986,037	87	239,134
180	4,893	3,248	1,043,012	87	246,024
185	5,028	3,338	1,101,587	87	252,914
190	5,163	3,428	1,161,762	87	259,804
195	5,298	3,518	1,223,537	87	266,694
200	5,433	3,608	1,286,912	87	273,584
$n > 2$	$27n + 33$	$18n + 8$	$32n^2 + 35n - 88$	87	$1378n - 2016$

Table 34: Size Information for **mutual exclusion**

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
2	87	42	114	6	139
3	114	60	255	6	221
4	141	78	466	6	303
5	168	96	741	6	385
6	195	114	1,080	6	467
7	222	132	1,483	6	549
8	249	150	1,950	6	631
9	276	168	2,481	6	713
10	303	186	3,076	6	795
15	438	276	7,011	6	1,205
20	573	366	12,546	6	1,615
25	708	456	19,681	6	2,025
30	843	546	28,416	6	2,435
35	978	636	38,751	6	2,845
40	1,113	726	50,686	6	3,255
45	1,248	816	64,221	6	3,665

Table 35: Size Information for **no fork 1 up twice**

Philosophers	LOC	Nodes	Edges	Tuples	Node-Tuples
50	1,383	906	79,356	6	4,075
55	1,518	996	96,091	6	4,485
60	1,653	1,086	114,426	6	4,895
65	1,788	1,176	134,361	6	5,305
70	1,923	1,266	155,896	6	5,715
75	2,058	1,356	179,031	6	6,125
80	2,193	1,446	203,766	6	6,535
85	2,328	1,536	230,101	6	6,945
90	2,463	1,626	258,036	6	7,355
95	2,598	1,716	287,571	6	7,765
100	2,733	1,806	318,706	6	8,175
105	2,868	1,896	351,441	6	8,585
110	3,003	1,986	385,776	6	8,995
115	3,138	2,076	421,711	6	9,405
120	3,273	2,166	459,246	6	9,815
125	3,408	2,256	498,381	6	10,225
130	3,543	2,346	539,116	6	10,635
135	3,678	2,436	581,451	6	11,045
140	3,813	2,526	625,386	6	11,455
145	3,948	2,616	670,921	6	11,865
150	4,083	2,706	718,056	6	12,275
155	4,218	2,796	766,791	6	12,685
160	4,353	2,886	817,126	6	13,095
165	4,488	2,976	869,061	6	13,505
170	4,623	3,066	922,596	6	13,915
175	4,758	3,156	977,731	6	14,325
180	4,893	3,246	1,034,466	6	14,735
185	5,028	3,336	1,092,801	6	15,145
190	5,163	3,426	1,152,736	6	15,555
195	5,298	3,516	1,214,271	6	15,965
200	5,433	3,606	1,277,406	6	16,375
$n > 2$	$27n + 33$	$18n + 6$	$32n^2 - 13n + 6$	6	$82n - 25$

Table 35: Size Information for **no fork 1 up twice**

B.4 Chiron

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
2	556	96	278	24	851
3	785	121	378	30	1,361
4	1,016	152	494	39	2,234
5	1,245	177	594	48	3,131
6	1,474	202	694	54	4,055
7	1,705	233	810	63	5,486
8	1,934	258	910	72	6,851
9	2,163	283	1,010	78	8,189
10	2,393	314	1,126	87	10,178

Table 36: Size Information for **p01**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
11	2,624	339	1,226	96	12,011
12	2,853	364	1,326	102	13,763
13	3,084	395	1,442	111	16,310
14	3,313	420	1,542	120	18,611
15	3,542	445	1,642	126	20,777
16	3,773	476	1,758	135	23,882
17	4,002	501	1,858	144	26,651
18	4,231	526	1,958	150	29,231
19	4,462	557	2,074	159	32,894
20	4,690	582	2,174	168	36,131
21	4,921	607	2,274	174	39,125
22	5,152	638	2,390	183	43,346
23	5,381	663	2,490	192	47,051
24	5,610	688	2,590	198	50,459
25	5,841	719	2,706	207	55,238
26	6,070	744	2,806	216	59,411
27	6,299	769	2,906	222	63,233
28	6,530	800	3,022	231	68,570
29	6,759	825	3,122	240	73,211
30	6,987	850	3,222	246	77,447
31	7,220	881	3,338	255	83,342
32	7,449	906	3,438	264	88,451
33	7,678	931	3,538	270	93,101
34	7,909	962	3,654	279	99,554
35	8,138	987	3,754	288	105,131
36	8,367	1,012	3,854	294	110,195
37	8,598	1,043	3,970	303	117,206
38	8,827	1,068	4,070	312	123,251
39	9,056	1,093	4,170	318	128,729
40	9,286	1,124	4,286	327	136,298
41	9,517	1,149	4,386	336	142,811
42	9,746	1,174	4,486	342	148,703
43	9,977	1,205	4,602	351	156,830
44	10,206	1,230	4,702	360	163,811
45	10,435	1,255	4,802	366	170,117
46	10,666	1,286	4,918	375	178,802
47	10,895	1,311	5,018	384	186,251
48	11,124	1,336	5,118	390	192,971
49	11,355	1,367	5,234	399	202,214
50	11,583	1,392	5,334	408	210,131
51	11,814	1,417	5,434	414	217,265
52	12,045	1,448	5,550	423	227,066
53	12,274	1,473	5,650	432	235,451
$n \bmod 3 \equiv 0$		$27n + 40$	$105n + 62$	$8n + 6$	$80n^2 + 178n + 107$
$n \bmod 3 \equiv 1$		$27n + 44$	$105n + 72$	$8n + 7$	$80n^2 + 204n + 138$
$n \bmod 3 \equiv 2$		$27n + 42$	$105n + 67$	$8n + 8$	$80n^2 + 200n + 131$

Table 36: Size Information for **p01**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
2	556	162	457	575	7,673
3	785	187	557	830	11,393
4	1,016	218	673	1,130	16,043
5	1,245	243	773	1,385	19,808
6	1,474	268	873	1,640	23,528
7	1,705	299	989	1,940	28,178
8	1,934	324	1,089	2,195	31,943
9	2,163	349	1,189	2,450	35,663
10	2,393	380	1,305	2,750	40,313
11	2,624	405	1,405	3,005	44,078
12	2,853	430	1,505	3,260	47,798
13	3,084	461	1,621	3,560	52,448
14	3,313	486	1,721	3,815	56,213
15	3,542	511	1,821	4,070	59,933
16	3,773	542	1,937	4,370	64,583
17	4,002	567	2,037	4,625	68,348
18	4,231	592	2,137	4,880	72,068
19	4,462	623	2,253	5,180	76,718
20	4,690	648	2,353	5,435	80,483
21	4,921	673	2,453	5,690	84,203
22	5,152	704	2,569	5,990	88,853
23	5,381	729	2,669	6,245	92,618
24	5,610	754	2,769	6,500	96,338
25	5,841	785	2,885	6,800	100,988
26	6,070	810	2,985	7,055	104,753
27	6,299	835	3,085	7,310	108,473
28	6,530	866	3,201	7,610	113,123
29	6,759	891	3,301	7,865	116,888
30	6,987	916	3,401	8,120	120,608
31	7,220	947	3,517	8,420	125,258
32	7,449	972	3,617	8,675	129,023
33	7,678	997	3,717	8,930	132,743
34	7,909	1,028	3,833	9,230	137,393
35	8,138	1,053	3,933	9,485	141,158
36	8,367	1,078	4,033	9,740	144,878
37	8,598	1,109	4,149	10,040	149,528
38	8,827	1,134	4,249	10,295	153,293
39	9,056	1,159	4,349	10,550	157,013
40	9,286	1,190	4,465	10,850	161,663
41	9,517	1,215	4,565	11,105	165,428
42	9,746	1,240	4,665	11,360	169,148
43	9,977	1,271	4,781	11,660	173,798
44	10,206	1,296	4,881	11,915	177,563
45	10,435	1,321	4,981	12,170	181,283
46	10,666	1,352	5,097	12,470	185,933
47	10,895	1,377	5,197	12,725	189,698
48	11,124	1,402	5,297	12,980	193,418

Table 37: Size Information for **p02**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
49	11,355	1,433	5,413	13,280	198,068
50	11,583	1,458	5,513	13,535	201,833
51	11,814	1,483	5,613	13,790	205,553
52	12,045	1,514	5,729	14,090	210,203
53	12,274	1,539	5,829	14,345	213,968
n mod $3 \equiv 0$		$27n + 106$	$105n + 241$	$270n + 20$	$4045n - 742$
n mod $3 \equiv 1$		$27n + 110$	$105n + 251$	$270n + 50$	$4045n - 137$
n mod $3 \equiv 2$		$27n + 108$	$105n + 246$	$270n + 35$	$4045n - 417$

Table 37: Size Information for **p02**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
2	556	96	278	63	974
3	785	121	378	97	1,470
4	1,016	152	494	137	2,090
5	1,245	177	594	171	2,592
6	1,474	202	694	205	3,088
7	1,705	233	810	245	3,708
8	1,934	258	910	279	4,210
9	2,163	283	1,010	313	4,706
10	2,393	314	1,126	353	5,326
11	2,624	339	1,226	387	5,828
12	2,853	364	1,326	421	6,324
13	3,084	395	1,442	461	6,944
14	3,313	420	1,542	495	7,446
15	3,542	445	1,642	529	7,942
16	3,773	476	1,758	569	8,562
17	4,002	501	1,858	603	9,064
18	4,231	526	1,958	637	9,560
19	4,462	557	2,074	677	10,180
20	4,690	582	2,174	711	10,682
21	4,921	607	2,274	745	11,178
22	5,152	638	2,390	785	11,798
23	5,381	663	2,490	819	12,300
24	5,610	688	2,590	853	12,796
25	5,841	719	2,706	893	13,416
26	6,070	744	2,806	927	13,918
27	6,299	769	2,906	961	14,414
28	6,530	800	3,022	1,001	15,034
29	6,759	825	3,122	1,035	15,536
30	6,987	850	3,222	1,069	16,032
31	7,220	881	3,338	1,109	16,652
32	7,449	906	3,438	1,143	17,154
33	7,678	931	3,538	1,177	17,650
34	7,909	962	3,654	1,217	18,270
35	8,138	987	3,754	1,251	18,772

Table 38: Size Information for **p03**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
36	8,367	1,012	3,854	1,285	19,268
37	8,598	1,043	3,970	1,325	19,888
38	8,827	1,068	4,070	1,359	20,390
39	9,056	1,093	4,170	1,393	20,886
40	9,286	1,124	4,286	1,433	21,506
41	9,517	1,149	4,386	1,467	22,008
42	9,746	1,174	4,486	1,501	22,504
43	9,977	1,205	4,602	1,541	23,124
44	10,206	1,230	4,702	1,575	23,626
45	10,435	1,255	4,802	1,609	24,122
46	10,666	1,286	4,918	1,649	24,742
47	10,895	1,311	5,018	1,683	25,244
48	11,124	1,336	5,118	1,717	25,740
49	11,355	1,367	5,234	1,757	26,360
50	11,583	1,392	5,334	1,791	26,862
51	11,814	1,417	5,434	1,825	27,358
52	12,045	1,448	5,550	1,865	27,978
53	12,274	1,473	5,650	1,899	28,480

Table 38: Size Information for **p03**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
2	556	96	278	49	785
3	785	121	378	83	1,281
4	1,016	152	494	123	1,901
5	1,245	177	594	157	2,403
6	1,474	202	694	191	2,899
7	1,705	233	810	231	3,519
8	1,934	258	910	265	4,021
9	2,163	283	1,010	299	4,517
10	2,393	314	1,126	339	5,137
11	2,624	339	1,226	373	5,639
12	2,853	364	1,326	407	6,135
13	3,084	395	1,442	447	6,755
14	3,313	420	1,542	481	7,257
15	3,542	445	1,642	515	7,753
16	3,773	476	1,758	555	8,373
17	4,002	501	1,858	589	8,875
18	4,231	526	1,958	623	9,371
19	4,462	557	2,074	663	9,991
20	4,690	582	2,174	697	10,493
21	4,921	607	2,274	731	10,989
22	5,152	638	2,390	771	11,609
23	5,381	663	2,490	805	12,111
24	5,610	688	2,590	839	12,607
25	5,841	719	2,706	879	13,227
26	6,070	744	2,806	913	13,729

Table 39: Size Information for **p04**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
27	6,299	769	2,906	947	14,225
28	6,530	800	3,022	987	14,845
29	6,759	825	3,122	1,021	15,347
30	6,987	850	3,222	1,055	15,843
31	7,220	881	3,338	1,095	16,463
32	7,449	906	3,438	1,129	16,965
33	7,678	931	3,538	1,163	17,461
34	7,909	962	3,654	1,203	18,081
35	8,138	987	3,754	1,237	18,583
36	8,367	1,012	3,854	1,271	19,079
37	8,598	1,043	3,970	1,311	19,699
38	8,827	1,068	4,070	1,345	20,201
39	9,056	1,093	4,170	1,379	20,697
40	9,286	1,124	4,286	1,419	21,317
41	9,517	1,149	4,386	1,453	21,819
42	9,746	1,174	4,486	1,487	22,315
43	9,977	1,205	4,602	1,527	22,935
44	10,206	1,230	4,702	1,561	23,437
45	10,435	1,255	4,802	1,595	23,933
46	10,666	1,286	4,918	1,635	24,553
47	10,895	1,311	5,018	1,669	25,055
48	11,124	1,336	5,118	1,703	25,551
49	11,355	1,367	5,234	1,743	26,171
50	11,583	1,392	5,334	1,777	26,673
51	11,814	1,417	5,434	1,811	27,169
52	12,045	1,448	5,550	1,851	27,789
53	12,274	1,473	5,650	1,885	28,291

Table 39: Size Information for **p04**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
2	556	96	278	425	4,511
3	785	121	378	761	9,198
4	1,016	152	494	1,301	18,561
5	1,245	177	594	1,844	29,924
6	1,474	202	694	2,486	44,218
7	1,705	233	810	3,404	68,057
8	1,934	258	910	4,253	93,517
9	2,163	283	1,010	5,201	122,554
10	2,393	314	1,126	6,497	167,445
11	2,624	339	1,226	7,652	212,570
12	2,853	364	1,326	8,906	261,486
13	3,084	395	1,442	10,580	334,005
14	3,313	420	1,542	12,041	404,363
15	3,542	445	1,642	13,601	478,294
16	3,773	476	1,758	15,653	585,017
17	4,002	501	1,858	17,420	686,176

Table 40: Size Information for **p05**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
18	4,231	526	1,958	19,286	790,258
19	4,462	557	2,074	21,716	937,761
20	4,690	582	2,174	23,789	1,075,289
21	4,921	607	2,274	25,961	1,214,658
22	5,152	638	2,390	28,769	1,409,517
23	5,381	663	2,490	31,148	1,588,982
24	5,610	688	2,590	33,626	1,768,774
25	5,841	719	2,706	36,812	2,017,565
26	6,070	744	2,806	39,497	2,244,535
27	6,299	769	2,906	42,281	2,469,886
28	6,530	800	3,022	45,845	2,779,185
29	6,759	825	3,122	48,836	3,059,228
30	6,987	850	3,222	51,926	3,335,274
31	7,220	881	3,338	55,868	3,711,657
32	7,449	906	3,438	59,165	4,050,341
33	7,678	931	3,538	62,561	4,382,218
34	7,909	962	3,654	66,881	4,832,261
35	8,138	987	3,754	70,484	5,235,154
36	8,367	1,012	3,854	74,186	5,627,998
37	8,598	1,043	3,970	78,884	6,158,277
38	8,827	1,068	4,070	82,793	6,630,947
39	9,056	1,093	4,170	86,801	7,089,894
40	9,286	1,124	4,286	91,877	7,706,985
41	9,517	1,149	4,386	96,092	8,255,000
42	9,746	1,174	4,486	100,406	8,785,186
43	9,977	1,205	4,602	105,860	9,495,665
44	10,206	1,230	4,702	110,381	10,124,593
45	10,435	1,255	4,802	115,001	10,731,154
46	10,666	1,286	4,918	120,833	11,541,597
47	10,895	1,311	5,018	125,660	12,257,006
48	11,124	1,336	5,118	130,586	12,945,078
49	11,355	1,367	5,234	136,796	13,862,061
50	11,583	1,392	5,334	141,929	14,669,519
51	11,814	1,417	5,434	147,161	15,444,238
52	12,045	1,448	5,550	153,749	16,474,337
53	12,274	1,473	5,650	159,188	17,379,412

Table 40: Size Information for **p05**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
2	556	96	278	24	851
3	785	121	378	30	1,361
4	1,016	152	494	39	2,234
5	1,245	177	594	48	3,131
6	1,474	202	694	54	4,055
7	1,705	233	810	63	5,486
8	1,934	258	910	72	6,851

Table 41: Size Information for **p06**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
9	2,163	283	1,010	78	8,189
10	2,393	314	1,126	87	10,178
11	2,624	339	1,226	96	12,011
12	2,853	364	1,326	102	13,763
13	3,084	395	1,442	111	16,310
14	3,313	420	1,542	120	18,611
15	3,542	445	1,642	126	20,777
16	3,773	476	1,758	135	23,882
17	4,002	501	1,858	144	26,651
18	4,231	526	1,958	150	29,231
19	4,462	557	2,074	159	32,894
20	4,690	582	2,174	168	36,131
21	4,921	607	2,274	174	39,125
22	5,152	638	2,390	183	43,346
23	5,381	663	2,490	192	47,051
24	5,610	688	2,590	198	50,459
25	5,841	719	2,706	207	55,238
26	6,070	744	2,806	216	59,411
27	6,299	769	2,906	222	63,233
28	6,530	800	3,022	231	68,570
29	6,759	825	3,122	240	73,211
30	6,987	850	3,222	246	77,447
31	7,220	881	3,338	255	83,342
32	7,449	906	3,438	264	88,451
33	7,678	931	3,538	270	93,101
34	7,909	962	3,654	279	99,554
35	8,138	987	3,754	288	105,131
36	8,367	1,012	3,854	294	110,195
37	8,598	1,043	3,970	303	117,206
38	8,827	1,068	4,070	312	123,251
39	9,056	1,093	4,170	318	128,729
40	9,286	1,124	4,286	327	136,298
41	9,517	1,149	4,386	336	142,811
42	9,746	1,174	4,486	342	148,703
43	9,977	1,205	4,602	351	156,830
44	10,206	1,230	4,702	360	163,811
45	10,435	1,255	4,802	366	170,117
46	10,666	1,286	4,918	375	178,802
47	10,895	1,311	5,018	384	186,251
48	11,124	1,336	5,118	390	192,971
49	11,355	1,367	5,234	399	202,214
50	11,583	1,392	5,334	408	210,131
51	11,814	1,417	5,434	414	217,265
52	12,045	1,448	5,550	423	227,066
53	12,274	1,473	5,650	432	235,451
$n \bmod 3 \equiv 0$		$27n + 40$	$105n + 62$	$8n + 6$	$80n^2 + 178n + 107$
$n \bmod 3 \equiv 1$		$27n + 44$	$105n + 72$	$8n + 7$	$80n^2 + 204n + 138$

Table 41: Size Information for **p06**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
$n \bmod 3 \equiv 2$		$27n + 42$	$105n + 67$	$8n + 8$	$80n^2 + 200n + 131$

Table 41: Size Information for **p06**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
2	556	147	425	354	4,927
3	785	172	525	507	7,159
4	1,016	203	641	687	9,949
5	1,245	228	741	840	12,208
6	1,474	253	841	993	14,440
7	1,705	284	957	1,173	17,230
8	1,934	309	1,057	1,326	19,489
9	2,163	334	1,157	1,479	21,721
10	2,393	365	1,273	1,659	24,511
11	2,624	390	1,373	1,812	26,770
12	2,853	415	1,473	1,965	29,002
13	3,084	446	1,589	2,145	31,792
14	3,313	471	1,689	2,298	34,051
15	3,542	496	1,789	2,451	36,283
16	3,773	527	1,905	2,631	39,073
17	4,002	552	2,005	2,784	41,332
18	4,231	577	2,105	2,937	43,564
19	4,462	608	2,221	3,117	46,354
20	4,690	633	2,321	3,270	48,613
21	4,921	658	2,421	3,423	50,845
22	5,152	689	2,537	3,603	53,635
23	5,381	714	2,637	3,756	55,894
24	5,610	739	2,737	3,909	58,126
25	5,841	770	2,853	4,089	60,916
26	6,070	795	2,953	4,242	63,175
27	6,299	820	3,053	4,395	65,407
28	6,530	851	3,169	4,575	68,197
29	6,759	876	3,269	4,728	70,456
30	6,987	901	3,369	4,881	72,688
31	7,220	932	3,485	5,061	75,478
32	7,449	957	3,585	5,214	77,737
33	7,678	982	3,685	5,367	79,969
34	7,909	1,013	3,801	5,547	82,759
35	8,138	1,038	3,901	5,700	85,018
36	8,367	1,063	4,001	5,853	87,250
37	8,598	1,094	4,117	6,033	90,040
38	8,827	1,119	4,217	6,186	92,299
39	9,056	1,144	4,317	6,339	94,531
40	9,286	1,175	4,433	6,519	97,321
41	9,517	1,200	4,533	6,672	99,580
42	9,746	1,225	4,633	6,825	101,812
43	9,977	1,256	4,749	7,005	104,602

Table 42: Size Information for **p07**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
44	10,206	1,281	4,849	7,158	106,861
45	10,435	1,306	4,949	7,311	109,093
46	10,666	1,337	5,065	7,491	111,883
47	10,895	1,362	5,165	7,644	114,142
48	11,124	1,387	5,265	7,797	116,374
49	11,355	1,418	5,381	7,977	119,164
50	11,583	1,443	5,481	8,130	121,423
51	11,814	1,468	5,581	8,283	123,655
52	12,045	1,499	5,697	8,463	126,445
53	12,274	1,524	5,797	8,616	128,704
$n \bmod 3 \equiv 0$		$27n + 91$	$105n + 209$	$162n + 21$	$2427n - 122$
$n \bmod 3 \equiv 1$		$27n + 95$	$105n + 219$	$162n + 39$	$2427n + 241$
$n \bmod 3 \equiv 2$		$27n + 93$	$105n + 214$	$162n + 30$	$2427n + 73$

Table 42: Size Information for **p07**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
2	556	164	459	575	7,673
3	785	189	559	830	11,393
4	1,016	220	675	1,130	16,043
5	1,245	245	775	1,385	19,808
6	1,474	270	875	1,640	23,528
7	1,705	301	991	1,940	28,178
8	1,934	326	1,091	2,195	31,943
9	2,163	351	1,191	2,450	35,663
10	2,393	382	1,307	2,750	40,313
11	2,624	407	1,407	3,005	44,078
12	2,853	432	1,507	3,260	47,798
13	3,084	463	1,623	3,560	52,448
14	3,313	488	1,723	3,815	56,213
15	3,542	513	1,823	4,070	59,933
16	3,773	544	1,939	4,370	64,583
17	4,002	569	2,039	4,625	68,348
18	4,231	594	2,139	4,880	72,068
19	4,462	625	2,255	5,180	76,718
20	4,690	650	2,355	5,435	80,483
21	4,921	675	2,455	5,690	84,203
22	5,152	706	2,571	5,990	88,853
23	5,381	731	2,671	6,245	92,618
24	5,610	756	2,771	6,500	96,338
25	5,841	787	2,887	6,800	100,988
26	6,070	812	2,987	7,055	104,753
27	6,299	837	3,087	7,310	108,473
28	6,530	868	3,203	7,610	113,123
29	6,759	893	3,303	7,865	116,888
30	6,987	918	3,403	8,120	120,608

Table 43: Size Information for **p08**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
31	7,220	949	3,519	8,420	125,258
32	7,449	974	3,619	8,675	129,023
33	7,678	999	3,719	8,930	132,743
34	7,909	1,030	3,835	9,230	137,393
35	8,138	1,055	3,935	9,485	141,158
36	8,367	1,080	4,035	9,740	144,878
37	8,598	1,111	4,151	10,040	149,528
38	8,827	1,136	4,251	10,295	153,293
39	9,056	1,161	4,351	10,550	157,013
40	9,286	1,192	4,467	10,850	161,663
41	9,517	1,217	4,567	11,105	165,428
42	9,746	1,242	4,667	11,360	169,148
43	9,977	1,273	4,783	11,660	173,798
44	10,206	1,298	4,883	11,915	177,563
45	10,435	1,323	4,983	12,170	181,283
46	10,666	1,354	5,099	12,470	185,933
47	10,895	1,379	5,199	12,725	189,698
48	11,124	1,404	5,299	12,980	193,418
49	11,355	1,435	5,415	13,280	198,068
50	11,583	1,460	5,515	13,535	201,833
51	11,814	1,485	5,615	13,790	205,553
52	12,045	1,516	5,731	14,090	210,203
53	12,274	1,541	5,831	14,345	213,968
n mod $3 \equiv 0$		$27n + 108$	$105n + 243$	$270n + 20$	$4045n - 742$
n mod $3 \equiv 1$		$27n + 112$	$105n + 253$	$270n + 50$	$4045n - 137$
n mod $3 \equiv 2$		$27n + 110$	$105n + 248$	$270n + 35$	$4045n - 417$

Table 43: Size Information for **p08**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
2	556	96	374	559	5,984
3	785	121	538	1,009	12,480
4	1,016	152	812	1,793	26,448
5	1,245	177	1,074	2,526	42,782
6	1,474	202	1,302	3,398	63,506
7	1,705	233	1,700	4,744	100,167
8	1,934	258	2,062	5,899	137,612
9	2,163	283	2,354	7,193	180,508
10	2,393	314	2,876	9,101	250,542
11	2,624	339	3,338	10,678	317,690
12	2,853	364	3,694	12,394	390,702
13	3,084	395	4,340	14,864	504,789
14	3,313	420	4,902	16,863	610,232
15	3,542	445	5,322	19,001	721,304
16	3,773	476	6,092	22,033	890,124
17	4,002	501	6,754	24,454	1,042,454

Table 44: Size Information for **p09**

Artists	LOC	Nodes	Edges	Tuples	Node-Tuples
18	4,231	526	7,238	27,014	1,199,530
19	4,462	557	8,132	30,608	1,433,763
20	4,690	582	8,894	33,451	1,641,572
21	4,921	607	9,442	36,433	1,852,596
22	5,152	638	10,460	40,589	2,162,922
23	5,381	663	11,322	43,854	2,434,802
24	5,610	688	11,934	47,258	2,707,718
25	5,841	719	13,076	51,976	3,104,817
26	6,070	744	14,038	55,663	3,449,360
27	6,299	769	14,714	59,489	3,792,112
28	6,530	800	15,980	64,769	4,286,664
29	6,759	825	17,042	68,878	4,712,462
30	6,987	850	17,782	73,126	5,132,994
31	7,220	881	19,172	78,968	5,735,679
32	7,449	906	20,334	83,499	6,251,324
33	7,678	931	21,138	88,169	6,757,580
34	7,909	962	22,652	94,573	7,479,078
35	8,138	987	23,914	99,526	8,093,162
36	8,367	1,012	24,782	104,618	8,693,086
37	8,598	1,043	26,420	111,584	9,544,077
38	8,827	1,068	27,782	116,959	10,265,192
39	9,056	1,093	28,714	122,473	10,966,728
40	9,286	1,124	30,476	130,001	11,957,892
41	9,517	1,149	31,938	135,798	12,794,630
42	9,746	1,174	32,934	141,734	13,605,722
43	9,977	1,205	34,820	149,824	14,747,739
44	10,206	1,230	36,382	156,043	15,708,692
45	10,435	1,255	37,442	162,401	16,637,284
46	10,666	1,286	39,452	171,053	17,940,834
47	10,895	1,311	41,114	177,694	19,034,594
48	11,124	1,336	42,238	184,474	20,088,630
49	11,355	1,367	44,372	193,688	21,564,393
50	11,583	1,392	46,134	200,751	22,799,552
51	11,814	1,417	47,322	207,953	23,986,976
52	12,045	1,448	49,580	217,729	25,645,632
53	12,274	1,473	51,442	225,214	27,030,782

Table 44: Size Information for **p09**

C Time Information

This section gives timing information for all of the properties presented in the body of the paper. All times are given in seconds. Since the FLAVERS/Ada toolset has been programmed in three different languages, we have broken down the timing information. The time for running the tools that construct CFGs from the Ada source code, written in Ada, are given in the "Language Processing" column. The time for running the tools, written in Java, that convert these CFGs into a TFG and construct the variable and property automata are given in the "Artifact Building" column. Finally, the times for the MHP analysis and state propagation, both written in C, are given in the "MHP" and "State Propagation" columns, respectively.

C.1 The Reader/Writer Program

Readers	Language Processing	Artifact Building	MHP	State Propagation
1	10.31	2.35	0.01	0.02
2	13.38	2.51	0.01	0.02
3	16.68	2.57	0.02	0.02
4	20.02	2.68	0.02	0.02
5	23.49	2.85	0.03	0.01
6	27.09	2.96	0.04	0.01
7	30.72	3.11	0.04	0.01
8	34.74	3.12	0.05	0.02
9	39.03	3.25	0.05	0.02
10	42.87	3.38	0.06	0.02
15	65.31	3.81	0.10	0.03
20	90.12	4.10	0.14	0.05
25	118.23	4.49	0.19	0.08
30	149.41	5.03	0.26	0.10
35	183.95	5.46	0.32	0.14
40	222.87	5.80	0.38	0.17
45	262.45	6.13	0.46	0.22
50	305.57	6.45	0.53	0.28
55	351.50	6.83	0.71	0.31
60	405.60	7.07	0.76	0.39
65	457.26	7.50	0.87	0.46
70	513.57	7.74	0.97	0.54
75	572.31	8.22	1.11	0.63
80	636.26	8.50	1.21	0.70
85	698.54	8.82	1.34	0.75
90	770.98	9.18	1.48	0.87
95	839.61	9.54	1.63	0.97
100	915.49	9.94	1.78	1.05
105	994.87	10.35	1.94	1.22
110	1,075.11	10.76	2.47	1.22
115	1,167.92	11.20	2.67	1.38
120	1,247.16	11.61	2.87	1.42
125	1,340.97	12.08	3.05	1.59
130	1,432.88	12.41	3.24	1.69
135	1,536.26	12.87	3.45	1.80
140	1,639.08	12.75	3.56	2.11
145	1,748.94	13.32	3.93	2.08
150	1,853.33	13.77	4.04	2.52
155	1,986.23	14.28	4.36	2.44
160	2,078.24	14.50	4.57	2.61
165	2,199.44	15.02	4.99	2.54
170	2,343.47	15.52	5.09	2.96
175	2,443.62	16.21	5.45	2.88
180	2,562.35	16.70	5.68	3.18
185	2,685.16	17.02	5.81	3.70

Table 45: Time Information for **exclusive read write**

Readers	Language Processing	Artifact Building	MHP	State Propagation
190	2,878.66	17.86	6.26	3.65
195	2,969.45	17.84	6.68	3.52
200	3,102.59	18.30	6.92	3.80

Table 45: Time Information for **exclusive read write**

Readers	Language Processing	Artifact Building	MHP	State Propagation
1	10.31	2.35	0.01	0.02
2	13.38	2.49	0.01	0.02
3	16.68	2.66	0.02	0.02
4	20.02	2.89	0.02	0.02
5	23.49	3.04	0.03	0.02
6	27.09	3.27	0.03	0.03
7	30.72	3.50	0.04	0.03
8	34.74	3.67	0.04	0.04
9	39.03	3.82	0.05	0.05
10	42.87	4.05	0.06	0.06
15	65.31	5.34	0.10	0.14
20	90.12	6.82	0.13	0.31
25	118.23	9.53	0.18	0.54
30	149.41	12.84	0.25	0.94
35	183.95	18.25	0.31	1.33
40	222.87	26.09	0.36	1.92
45	262.45	35.98	0.43	2.69
50	305.57	49.11	0.55	3.54
55	351.50	65.27	0.68	4.68
60	405.60	87.40	0.75	5.99
65	457.26	115.23	0.94	7.39
70	513.57	147.27	0.99	9.19
75	572.31	185.05	1.12	11.23
80	636.26	234.64	1.18	13.55
85	698.54	287.35	1.33	16.14
90	770.98	355.93	1.47	19.04
95	839.61	431.84	1.57	22.45
100	915.49	528.09	1.71	26.27
105	994.87	634.79	1.89	30.01
110	1,075.11	759.27	2.39	34.19
115	1,167.92	891.26	2.54	39.07
120	1,247.16	1,031.15	2.74	44.55
125	1,340.97	1,227.84	2.96	49.74
130	1,432.88	1,423.26	3.12	55.98
135	1,536.26	1,659.95	3.35	62.43
140	1,639.08	1,898.97	3.54	69.53
145	1,748.94	2,176.03	3.78	76.78
150	1,853.33	2,491.65	3.98	85.72
155	1,986.23	2,790.86	4.30	93.24
160	2,078.24	3,165.22	4.45	102.41

Table 46: Time Information for **no write while reading**

Readers	Language Processing	Artifact Building	MHP	State Propagation
165	2,199.44	3,482.43	4.86	111.95
170	2,343.47	3,909.01	5.08	121.84
175	2,443.62	4,104.39	5.34	133.48
180	2,562.35	4,552.48	5.68	144.97
185	2,685.16	5,007.25	5.78	157.43
190	2,878.66	5,549.62	6.09	172.47
195	2,969.45	6,155.09	6.44	185.80
200	3,102.59	6,694.03	6.64	200.69

Table 46: Time Information for **no write while reading**

Readers	Language Processing	Artifact Building	MHP	State Propagation
1	10.31	2.28	0.01	0.02
2	13.38	2.41	0.01	0.02
3	16.68	2.51	0.02	0.02
4	20.02	2.61	0.02	0.02
5	23.49	2.75	0.03	0.02
6	27.09	2.87	0.03	0.02
7	30.72	3.01	0.04	0.01
8	34.74	3.05	0.04	0.02
9	39.03	3.18	0.05	0.02
10	42.87	3.22	0.06	0.02
15	65.31	3.63	0.10	0.04
20	90.12	4.02	0.14	0.06
25	118.23	4.34	0.18	0.09
30	149.41	4.91	0.24	0.14
35	183.95	5.22	0.30	0.19
40	222.87	5.62	0.36	0.25
45	262.45	5.94	0.44	0.28
50	305.57	6.33	0.52	0.35
55	351.50	6.67	0.68	0.46
60	405.60	7.17	0.76	0.50
65	457.26	7.28	0.87	0.58
70	513.57	7.73	0.95	0.68
75	572.31	8.05	1.09	0.78
80	636.26	8.45	1.19	0.89
85	698.54	8.79	1.35	0.94
90	770.98	9.23	1.45	1.14
95	839.61	9.56	1.56	1.28
100	915.49	9.93	1.74	1.36
105	994.87	10.28	1.88	1.59
110	1,075.11	10.58	2.43	1.56
115	1,167.92	11.15	2.60	1.74
120	1,247.16	11.35	2.77	1.89
125	1,340.97	11.79	3.01	2.06
130	1,432.88	12.16	3.17	2.22
135	1,536.26	12.60	3.38	2.44

Table 47: Time Information for **no write while reading 2**

Readers	Language Processing	Artifact Building	MHP	State Propagation
140	1,639.08	12.96	3.53	2.69
145	1,748.94	13.26	3.81	2.80
150	1,853.33	13.83	4.02	3.01
155	1,986.23	13.97	4.28	3.16
160	2,078.24	14.44	4.50	3.37
165	2,199.44	15.22	4.82	3.54
170	2,343.47	15.60	5.16	3.70
175	2,443.62	15.90	5.24	4.09
180	2,562.35	16.22	5.54	4.30
185	2,685.16	16.70	5.87	4.51
190	2,878.66	17.30	6.14	4.97
195	2,969.45	17.82	6.61	4.78
200	3,102.59	18.01	6.62	5.36

Table 47: Time Information for **no write while reading 2**

Readers	Language Processing	Artifact Building	MHP	State Propagation
1	10.31	2.28	0.01	0.02
2	13.38	2.41	0.01	0.02
3	16.68	2.45	0.02	0.02
4	20.02	2.55	0.02	0.02
5	23.49	2.74	0.02	0.02
6	27.09	2.84	0.03	0.01
7	30.72	2.90	0.04	0.01
8	34.74	3.08	0.04	0.02
9	39.03	3.17	0.05	0.02
10	42.87	3.17	0.05	0.03
15	65.31	3.64	0.09	0.06
20	90.12	3.92	0.13	0.08
25	118.23	4.28	0.17	0.13
30	149.41	4.62	0.23	0.21
35	183.95	5.11	0.29	0.26
40	222.87	5.50	0.34	0.35
45	262.45	6.07	0.43	0.39
50	305.57	6.40	0.50	0.51
55	351.50	6.87	0.63	0.61
60	405.60	7.02	0.72	0.70
65	457.26	7.54	0.81	0.82
70	513.57	7.79	0.90	0.95
75	572.31	8.37	0.99	1.14
80	636.26	8.62	1.11	1.29
85	698.54	9.19	1.23	1.44
90	770.98	9.61	1.37	1.68
95	839.61	9.82	1.51	1.85
100	915.49	10.29	1.62	2.00
105	994.87	10.74	1.75	2.25
110	1,075.11	11.12	2.23	2.42

Table 48: Time Information for **write first**

Readers	Language Processing	Artifact Building	MHP	State Propagation
115	1,167.92	11.30	2.44	2.54
120	1,247.16	12.00	2.58	2.85
125	1,340.97	12.17	2.80	3.09
130	1,432.88	12.78	2.98	3.32
135	1,536.26	12.90	3.15	3.61
140	1,639.08	13.60	3.29	3.93
145	1,748.94	14.21	3.58	4.19
150	1,853.33	14.57	3.78	4.50
155	1,986.23	14.97	4.02	4.86
160	2,078.24	15.25	4.19	7.06
165	2,199.44	16.07	4.48	5.29
170	2,343.47	16.36	4.81	5.58
175	2,443.62	16.73	4.91	6.04
180	2,562.35	17.24	5.22	6.36
185	2,685.16	17.55	5.54	6.71
190	2,878.66	18.15	5.89	9.66
195	2,969.45	18.51	6.26	7.15
200	3,102.59	18.94	6.29	7.91

Table 48: Time Information for **write first**

C.2 The Gas Station Program

Customers	Language Processing	Artifact Building	MHP	State Propagation
2	13.20	2.50	0.02	0.02
3	15.89	2.55	0.03	0.12
4	18.88	2.70	0.04	1.06
5	22.23	2.84	0.05	8.43
6	25.37	3.02	0.07	64.50
7	29.00	3.10	0.07	465.08

Table 49: Time Information for **mutual exclusion**, 1 pump

Customers	Language Processing	Artifact Building	MHP	State Propagation
2	17.16	2.66	0.04	0.27
3	21.59	2.96	0.07	8.51
4	26.25	3.20	0.09	210.40
5	31.32	3.50	0.11	4,440.17

Table 50: Time Information for **mutual exclusion**, 2 pumps

Customers	Language Processing	Artifact Building	MHP	State Propagation
2	21.78	2.97	0.06	1.14
3	27.78	3.24	0.10	66.36
4	34.77	3.58	0.15	2,752.95

Table 51: Time Information for **mutual exclusion**, 3 pumps

Customers	Language Processing	Artifact Building	MHP	State Propagation
1	10.66	2.26	0.01	0.02
2	13.20	2.42	0.02	0.02
3	15.89	2.52	0.03	0.02
4	18.88	2.76	0.04	0.02
5	22.23	2.84	0.05	0.02
6	25.37	2.96	0.06	0.01
7	29.00	3.12	0.07	0.02
8	32.80	3.23	0.08	0.02
9	36.72	3.39	0.10	0.02
10	40.96	3.42	0.11	0.03
15	65.46	4.07	0.20	0.03
20	95.51	4.77	0.30	0.06
25	130.73	5.32	0.42	0.10
30	170.72	6.09	0.56	0.14
35	217.71	6.70	0.73	0.19
40	269.89	7.24	0.90	0.25
45	329.50	8.07	1.10	0.32
50	392.99	8.74	1.37	0.38
55	457.82	9.49	1.61	0.47
60	536.42	10.25	1.87	0.55
65	622.55	10.86	2.19	0.65
70	701.15	12.00	2.52	0.66
75	791.52	12.88	2.91	0.75
80	886.84	14.06	3.34	0.87
85	998.03	14.91	3.75	0.97
90	1,106.96	16.50	4.08	1.17
95	1,220.08	17.03	4.54	1.31
100	1,341.87	17.70	4.91	1.49
105	1,458.41	18.85	5.58	1.63
110	1,596.75	19.87	6.23	1.65
115	1,747.86	21.26	6.72	2.05
120	1,895.65	21.47	7.40	2.17
125	2,024.84	23.69	7.97	2.42
130	2,191.92	24.76	8.81	2.29
135	2,357.15	26.33	9.48	2.42
140	2,515.85	27.07	9.98	2.78
145	2,701.50	28.21	11.06	2.82
150	2,879.06	28.79	11.62	3.17
155	3,086.53	30.38	12.38	3.56
160	3,259.17	31.26	13.10	3.93
165	3,431.91	32.99	14.08	3.86
170	3,661.25	33.92	14.70	4.32
175	3,863.44	34.88	15.98	4.09
180	4,075.99	37.11	16.54	4.71
185	4,321.58	37.44	17.60	5.09
190	4,521.09	39.67	18.56	5.18
195	4,843.94	40.99	19.60	5.10

Table 52: Time Information for **customer 1 start/stop**, 1 pump

Customers	Language Processing	Artifact Building	MHP	State Propagation
200	4,975.10	42.04	20.25	5.78

Table 52: Time Information for **customer 1 start/stop**, 1 pump

Customers	Language Processing	Artifact Building	MHP	State Propagation
1	13.33	2.48	0.02	0.02
2	17.16	2.70	0.04	0.02
3	21.59	2.90	0.06	0.01
4	26.25	3.15	0.09	0.01
5	31.32	3.32	0.11	0.03
6	36.76	3.45	0.14	0.04
7	42.86	3.79	0.18	0.04
8	49.32	3.99	0.21	0.06
9	55.83	4.23	0.25	0.08
10	63.40	4.44	0.29	0.11
15	106.31	5.66	0.56	0.22
20	160.31	6.76	0.91	0.37
25	226.35	7.98	1.36	0.60
30	303.19	9.15	1.90	0.84
35	390.09	10.34	2.52	1.14
40	491.81	12.63	3.41	1.40
45	603.49	14.14	4.13	1.93
50	723.02	16.03	5.12	2.34
55	860.46	17.60	6.23	2.74
60	1,005.25	19.55	7.28	3.38
65	1,166.14	21.21	8.55	4.01
70	1,341.85	24.02	9.99	4.60
75	1,527.55	26.75	11.31	5.31
80	1,710.62	29.31	12.80	6.12
85	1,921.73	31.50	14.67	6.58
90	2,146.59	32.26	16.11	7.65
95	2,360.47	36.66	18.07	8.36
100	2,605.06	39.02	20.00	9.29
105	2,873.67	41.21	22.06	10.39
110	3,163.82	44.38	24.19	11.28
115	3,438.75	48.45	26.73	12.03
120	3,712.83	50.24	28.85	13.29
125	4,009.03	53.83	31.33	14.26
130	4,361.13	56.84	34.29	15.46
135	4,684.45	60.81	36.51	17.46
140	5130.00	62.29	39.37	18.46
145	5,346.14	65.21	42.79	19.37
150	5,717.29	69.40	44.95	20.92
155	6,136.35	72.37	48.71	22.19
160	6,525.49	76.57	51.95	23.40
165	6,919.65	78.04	54.84	25.40
170	7,372.17	81.05	58.78	26.27

Table 53: Time Information for **customer 1 start/stop**, 2 pumps

Customers	Language Processing	Artifact Building	MHP	State Propagation
175	7,754.29	86.43	62.89	27.68
180	8,245.44	91.38	66.40	29.45
185	8,743.03	95.20	70.04	31.74
190	9,246.77	100.08	74.40	33.80
195	9,699.38	103.54	76.85	36.09
200	10,226.35	108.33	82.02	38.18

Table 53: Time Information for **customer 1 start/stop**, 2 pumps

Customers	Language Processing	Artifact Building	MHP	State Propagation
1	16.23	2.55	0.03	0.01
2	21.78	2.93	0.06	0.01
3	27.78	3.20	0.10	0.02
4	34.77	3.49	0.14	0.03
5	41.85	3.75	0.19	0.06
6	50.16	4.16	0.25	0.08
7	59.04	4.48	0.31	0.11
8	68.57	4.73	0.38	0.16
9	79.14	5.07	0.47	0.16
10	90.60	5.52	0.55	0.23
15	156.84	7.13	1.11	0.52
20	241.25	8.82	1.87	0.88
25	343.79	10.82	2.83	1.39
30	466.51	13.37	3.94	2.13
35	603.94	16.25	5.46	2.73
40	767.11	18.58	7.04	3.66
45	943.32	21.09	8.92	4.64
50	1,139.82	24.60	11.02	5.91
55	1,354.07	30.15	13.19	7.14
60	1,603.95	31.22	15.77	8.25
65	1,851.98	34.91	18.64	9.60
70	2,115.74	40.41	21.45	11.30
75	2,440.13	45.06	24.69	12.77
80	2,755.38	47.22	27.83	14.71
85	3,052.55	54.90	31.89	16.40
90	3,432.07	59.29	35.49	18.89
95	3,819.72	67.61	39.76	20.81
100	4,203.09	70.39	43.65	23.08
105	4,632.99	78.66	49.37	24.81
110	5,095.32	78.46	54.45	27.24
115	5,548.51	82.97	58.88	29.89
120	6,031.49	84.57	64.17	33.65
125	6,579.23	92.52	69.98	35.26
130	7,088.06	100.51	75.09	41.33
135	7,605.84	109.27	82.98	43.89
140	8,223.08	114.82	88.05	45.58
145	8,838.32	118.58	94.63	48.92

Table 54: Time Information for **customer 1 start/stop**, 3 pumps

Customers	Language Processing	Artifact Building	MHP	State Propagation
150	9,481.54	129.09	103.79	48.97
155	10,108.83	134.26	110.95	54.61
160	10,737.75	141.39	117.07	61.13
165	11353.00	149.50	126.26	62.89
170	12,073.26	159.48	134.77	67.41
175	12,840.67	169.40	142.84	72.20
180	13,587.33	173.90	150.86	75.96
185	14,397.05	183.22	159.70	79.53
190	15,233.21	192.22	171.44	82.10
195	16,068.61	200.69	178.95	88.75
200	16,834.51	215.52	190.67	91.93

Table 54: Time Information for **customer 1 start/stop**, 3 pumps

Customers	Language Processing	Artifact Building	MHP	State Propagation
1	10.66	2.27	0.01	0.02
2	13.20	2.46	0.02	0.02
3	15.89	2.50	0.03	0.02
4	18.88	2.66	0.04	0.02
5	22.23	2.78	0.05	0.04
6	25.37	2.98	0.06	0.06
7	29.00	3.13	0.07	0.09
8	32.80	3.26	0.08	0.14
9	36.72	3.38	0.10	0.18
10	40.96	3.41	0.11	0.23
15	65.46	3.96	0.19	0.71
20	95.51	4.81	0.29	1.58
25	130.73	5.45	0.41	2.95
30	170.72	6.08	0.56	4.97
35	217.71	6.81	0.74	7.79
40	269.89	7.33	0.93	11.38
45	329.50	8.30	1.15	16.36
50	392.99	8.80	1.37	22.23
55	457.82	9.64	1.65	28.84
60	536.42	10.68	1.98	37.26
65	622.55	10.97	2.28	47.09
70	701.15	12.04	2.59	58.61
75	791.52	12.85	2.98	72.00
80	886.84	14.14	3.41	86.97
85	998.03	15.22	3.84	103.93
90	1,106.96	16.26	4.31	123.97
95	1,220.08	16.88	4.79	144.69
100	1,341.87	17.70	5.19	168.19
105	1,458.41	19.01	5.75	195.30
110	1,596.75	20.21	6.30	223.56
115	1,747.86	21.19	6.91	255.60
120	1,895.65	21.86	7.44	292.72

Table 55: Time Information for **pump 1 start/stop**, 1 pump

Customers	Language Processing	Artifact Building	MHP	State Propagation
125	2,024.84	23.88	8.15	328.20
130	2,191.92	24.79	8.84	367.99
135	2,357.15	25.72	9.59	414.26
140	2,515.85	27.39	10.24	460.34
145	2,701.50	28.06	10.98	508.98
150	2,879.06	29.10	11.66	561.25
155	3,086.53	30.17	12.69	620.38
160	3,259.17	31.42	13.57	682.25
165	3,431.91	32.70	14.31	759.99
170	3,661.25	34.08	15.01	817.96
175	3,863.44	35.34	15.79	892.83
180	4,075.99	36.80	16.77	972.94
185	4,321.58	38.20	17.85	1,060.58
190	4,521.09	39.49	18.63	1,141.90
195	4,843.94	41.41	19.80	1,238.16
200	4,975.10	42.40	20.85	1,335.93

Table 55: Time Information for **pump 1 start/stop**, 1 pump

Customers	Language Processing	Artifact Building	MHP	State Propagation
1	13.33	2.44	0.02	0.02
2	17.16	2.64	0.05	0.02
3	21.59	2.93	0.06	0.09
4	26.25	3.10	0.08	0.19
5	31.32	3.30	0.11	0.34
6	36.76	3.52	0.14	0.57
7	42.86	3.79	0.18	0.88
8	49.32	3.92	0.22	1.29
9	55.83	4.22	0.27	1.80
10	63.40	4.45	0.30	2.47
15	106.31	5.66	0.58	8.08
20	160.31	6.66	0.95	18.78
25	226.35	7.94	1.44	36.31
30	303.19	9.23	1.99	62.44
35	390.09	10.71	2.73	98.74
40	491.81	12.46	3.48	147.12
45	603.49	14.26	4.48	217.09
50	723.02	15.92	5.48	285.73
55	860.46	18.67	6.61	380.31
60	1,005.25	19.65	7.72	494.78
65	1,166.14	21.56	9.02	646.35
70	1,341.85	24.04	10.58	793.66
75	1,527.55	27.01	12.04	980.67
80	1,710.62	29.38	13.65	1,166.69
85	1,921.73	31.45	15.58	1,418.07
90	2,146.59	32.69	17.41	1,664.53
95	2,360.47	36.55	19.29	1,955.72

Table 56: Time Information for **pump 1 start/stop**, 2 pumps

Customers	Language Processing	Artifact Building	MHP	State Propagation
100	2,605.06	39.12	21.66	2,281.90
105	2,873.67	41.14	23.91	2,637.48
110	3,163.82	44.89	25.88	3,024.46
115	3,438.75	48.40	28.36	3,610.88
120	3,712.83	50.63	30.77	4,092.11
125	4,009.03	52.31	33.98	4,460.43
130	4,361.13	55.40	36.54	5,018.03
135	4,684.45	60.46	39.41	5,615.26
140	5130.00	62.04	42.15	6,256.09
145	5,346.14	64.70	45.18	6,961.22
150	5,717.29	69.68	49.28	7,714.82
155	6,136.35	72.45	52.43	8,505.46
160	6,525.49	77.50	52.97	9,318.79
165	6,919.65	77.37	55.27	10,310.63
170	7,372.17	81.27	59.54	11,185.78
175	7,754.29	86.41	62.24	12,237.85
180	8,245.44	90.48	66.61	13,341.37
185	8,743.03	93.81	71.18	14,604.04
190	9,246.77	101.84	75.54	15,621.02
195	9,699.38	104.39	79.87	16,913.81
200	10,226.35	110.61	85.50	18,230.31

Table 56: Time Information for **pump 1 start/stop**, 2 pumps

Customers	Language Processing	Artifact Building	MHP	State Propagation
1	16.23	2.55	0.03	0.01
2	21.78	2.90	0.06	0.06
3	27.78	3.22	0.10	0.22
4	34.77	3.49	0.15	0.50
5	41.85	3.81	0.20	0.98
6	50.16	4.20	0.26	1.69
7	59.04	4.47	0.32	2.65
8	68.57	4.71	0.40	4.00
9	79.14	5.08	0.48	6.31
10	90.60	5.60	0.58	7.78
15	156.84	6.99	1.19	25.85
20	241.25	8.71	1.99	61.75
25	343.79	10.91	3.06	120.15
30	466.51	13.62	4.21	206.58
35	603.94	16.03	5.79	327.85
40	767.11	18.47	7.70	493.18
45	943.32	21.26	9.74	709.00
50	1,139.82	24.40	11.84	959.21
55	1,354.07	30.61	14.36	1,284.50
60	1,603.95	31.52	17.10	1,661.57
65	1,851.98	35.04	20.16	2,105.62
70	2,115.74	40.24	23.20	2,637.58

Table 57: Time Information for **pump 1 start/stop**, 3 pumps

Customers	Language Processing	Artifact Building	MHP	State Propagation
75	2,440.13	45.14	26.54	3,243.22
80	2,755.38	48.20	30.27	3,970.98
85	3,052.55	56.35	34.20	4,782.74
90	3,432.07	60.71	38.86	5,604.50
95	3,819.72	66.66	42.79	6,631.36
100	4,203.09	70.38	49.94	7,709.71
105	4,632.99	74.96	52.98	8,942.53
110	5,095.32	78.92	57.94	10,325.71
115	5,548.51	83.73	64.31	11,801.37
120	6,031.49	86.47	70.24	13,318.92
125	6,579.23	92.96	76.69	15,064.93
130	7,088.06	100.70	82.08	16,964.99
135	7,605.84	108.72	89.58	18,970.21
140	8,223.08	113.23	96.12	21,216.79
145	8,838.32	122.72	98.00	23,447.59
150	9,481.54	127.46	105.76	26,018.16
155	10,108.83	134.17	112.27	28,868.29
160	10,737.75	141.46	120.72	31,713.96
165	11353.00	148.62	132.15	34,744.02
170	12,073.26	159.13	137.49	38,339.41
175	12,840.67	168.50	147.82	41,431.76
180	13,587.33	176.43	157.53	45,219.58
185	14,397.05	183.39	163.80	48,848.31
190	15,233.21	195.22	176.63	52,944.06
195	16,068.61	201.98	185.95	57,241.29
200	16,834.51	211.80	195.34	61,614.47

Table 57: Time Information for **pump 1 start/stop**, 3 pumps

Customers	Language Processing	Artifact Building	MHP	State Propagation
1	13.33	2.47	0.02	0.02
2	17.16	2.64	0.04	0.17
3	21.59	2.94	0.06	1.61
4	26.25	3.14	0.08	6.13
5	31.32	3.36	0.11	16.70
6	36.76	3.53	0.14	36.83
7	42.86	3.72	0.18	71.79
8	49.32	3.86	0.21	126.92
9	55.83	4.24	0.25	209.61
10	63.40	4.55	0.29	326.90
15	106.31	5.64	0.54	1,775.73
20	160.31	6.88	0.93	5,872.42
25	226.35	7.78	1.33	14,822.10
30	303.19	9.32	1.84	31,580.67
35	390.09	10.65	2.50	59,339.27
40	491.81	12.36	3.15	102,732.63
45	603.49	14.18	4.03	167,096.91

Table 58: Time Information for **correct change**, 2 pumps

Customers	Language Processing	Artifact Building	MHP	State Propagation
46	628.60	14.51	4.23	182,531.10

Table 58: Time Information for **correct change**, 2 pumps

Customers	Language Processing	Artifact Building	MHP	State Propagation
1	16.23	2.56	0.03	0.01
2	21.78	3.00	0.06	1.00
3	27.78	3.24	0.10	15.53
4	34.77	3.46	0.14	63.86
5	41.85	3.78	0.19	180.89
6	50.16	4.13	0.24	418.78
7	59.04	4.47	0.31	827.70
8	68.57	4.76	0.38	1,483.82
9	79.14	5.14	0.45	2,483.08
10	90.60	5.59	0.54	3,915.99
15	156.84	7.18	1.08	22,280.86
20	241.25	8.83	1.84	74,946.24
25	343.79	10.88	2.74	190,029.58

Table 59: Time Information for **correct change**, 3 pumps

C.3 The Dining Philosophers Program

C.3.1 Standard

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
3	14.01	2.34	0.02	0.02
4	17.14	2.50	0.03	0.03
5	20.47	2.50	0.02	0.08
6	24.10	2.68	0.03	0.10
7	27.64	2.77	0.03	0.16
8	31.59	2.89	0.04	0.21
9	35.36	2.97	0.05	0.27
10	39.98	3.08	0.06	0.35
15	62.05	3.50	0.09	0.90
20	87.89	3.83	0.14	1.63
25	118.04	4.31	0.18	2.71
30	150.22	4.55	0.25	4.07
35	186.95	5.08	0.33	5.49
40	228.10	5.36	0.40	7.20
45	270.73	5.69	0.48	9.21
50	318.73	6.14	0.56	11.54
55	370.94	6.59	0.73	14.01
60	424.11	6.89	0.83	16.81
65	483.60	7.23	0.95	20.19
70	544.84	7.48	1.10	23.39

Table 60: Time Information for **mutual exclusion**

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
75	607.22	7.91	1.22	26.63
80	675.27	8.33	1.40	30.20
85	750.58	8.74	1.49	34.10
90	829.82	9.07	1.65	38.43
95	913.28	9.55	1.80	43.27
100	990.11	9.64	2.03	47.38
105	1,084.81	10.08	2.21	53.09
110	1,169.21	10.48	2.72	57.91
115	1,263.12	10.74	2.86	63.60
120	1,361.25	11.25	3.13	70.20
125	1,466.64	11.38	3.46	74.93
130	1,588.23	11.86	3.52	81.83
135	1,690.07	12.27	3.89	87.79
140	1,801.79	12.46	4.15	95.23
145	1,964.70	13.92	4.61	103.53
150	2,109.22	14.71	4.86	109.81
155	2,168.07	14.65	4.94	119.83
160	2,288.95	15.20	5.29	124.76
165	2,429.48	15.30	5.64	131.76
170	2,567.83	15.91	6.04	140.19
175	2,713.35	16.13	6.21	149.86
180	2,861.87	16.40	6.73	157.67
185	2,991.16	17.23	6.85	167.47
190	3,151.07	17.42	7.29	176.04
195	3,308.33	17.85	7.72	186.79
200	3,466.34	18.47	8.04	194.46

Table 60: Time Information for **mutual exclusion**

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
3	14.01	2.37	0.02	0.02
4	17.14	2.40	0.02	0.02
5	20.47	2.54	0.02	0.02
6	24.10	2.66	0.03	0.01
7	27.64	2.76	0.04	0.02
8	31.59	2.83	0.04	0.04
9	35.36	2.95	0.04	0.05
10	39.98	3.09	0.05	0.06
15	62.05	3.41	0.09	0.13
20	87.89	3.71	0.13	0.23
25	118.04	4.15	0.18	0.36
30	150.22	4.43	0.24	0.55
35	186.95	4.88	0.31	0.73
40	228.10	5.26	0.38	0.98
45	270.73	5.62	0.47	1.21
50	318.73	6.06	0.56	1.54
55	370.94	6.48	0.71	1.83

Table 61: Time Information for **no fork 1 up twice**

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
60	424.11	6.91	0.82	2.24
65	483.60	7.10	0.93	2.60
70	544.84	7.50	1.03	3.03
75	607.22	7.79	1.20	3.36
80	675.27	8.34	1.32	3.88
85	750.58	8.66	1.44	4.46
90	829.82	9.05	1.59	4.97
95	913.28	9.22	1.77	5.56
100	990.11	9.47	2.00	6.17
105	1,084.81	9.95	2.16	6.81
110	1,169.21	10.28	2.64	7.45
115	1,263.12	10.66	2.84	8.33
120	1,361.25	11.30	3.09	8.97
125	1,466.64	11.70	3.34	9.70
130	1,588.23	11.83	3.55	10.53
135	1,690.07	12.17	3.77	11.25
140	1,801.79	12.59	4.05	12.18
145	1,964.70	13.33	4.32	12.94
150	2,109.22	14.19	4.59	13.95
155	2,168.07	14.54	4.97	15.03
160	2,288.95	14.71	5.15	16.09
165	2,429.48	15.09	5.55	16.79
170	2,567.83	15.44	5.84	17.81
175	2,713.35	15.93	6.16	19.11
180	2,861.87	16.49	6.47	20.55
185	2,991.16	16.88	6.84	21.16
190	3,151.07	17.21	7.12	22.36
195	3,308.33	17.62	7.60	23.70
200	3,466.34	18.06	7.81	25.01

Table 61: Time Information for **no fork 1 up twice**

C.3.2 Fork Manager

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
2	10.57	2.37	0.02	0.02
3	12.78	2.48	0.02	0.02
4	15.06	2.53	0.02	0.02
5	17.47	2.62	0.01	0.03
6	20.21	2.71	0.02	0.03
7	22.86	2.73	0.02	0.04
8	25.56	2.85	0.02	0.06
9	28.53	2.90	0.02	0.07
10	31.84	3.02	0.03	0.09
15	49.30	3.37	0.04	0.19
20	69.99	3.67	0.05	0.34
25	95.14	3.96	0.07	0.52

Table 62: Time Information for **mutual exclusion**

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
30	122.84	4.36	0.08	0.75
35	153.69	4.59	0.10	1.00
40	188.82	4.99	0.11	1.32
45	227.54	5.24	0.14	1.68
50	268.92	5.55	0.16	2.12
55	316.25	5.79	0.18	2.49
60	365.58	6.17	0.21	2.95
65	420.03	6.36	0.24	3.58
70	475.78	6.62	0.26	4.01
75	536.58	6.95	0.29	4.62
80	599.24	7.41	0.32	5.25
85	665.07	7.70	0.35	5.91
90	737.27	8.04	0.39	6.63
95	808.87	8.12	0.42	7.49
100	891.40	8.43	0.46	8.15
105	973.23	8.75	0.49	9.05
110	1,050.41	9.22	0.54	9.90
115	1,151.94	9.46	0.57	10.81
120	1,247.31	9.78	0.60	11.85
125	1,343.68	10.09	0.66	12.78
130	1,440.24	10.43	0.74	13.91
135	1,549.18	10.64	0.79	14.90
140	1,658.09	11.11	0.82	16.12
145	1,781.20	11.22	0.88	17.25
150	1,915.78	12.33	0.97	18.80
155	1,997.38	11.93	0.95	19.74
160	2,128.31	12.29	1.00	20.97
165	2,257.28	12.66	1.10	23.89
170	2,375.21	12.69	1.14	23.74
175	2,506.76	13.44	1.18	25.01
180	2,656.16	13.59	1.26	26.62
185	2,814.12	13.55	1.32	27.93
190	2,952.26	14.55	1.38	29.57
195	3,099.22	14.79	1.48	30.96
200	3,253.16	14.72	1.45	33.05

Table 62: Time Information for **mutual exclusion**

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
2	10.57	2.35	0.01	0.02
3	12.78	2.44	0.01	0.02
4	15.06	2.48	0.02	0.02
5	17.47	2.61	0.02	0.02
6	20.21	2.67	0.02	0.02
7	22.86	2.71	0.03	0.02
8	25.56	2.83	0.02	0.02
9	28.53	2.93	0.02	0.02

Table 63: Time Information for **no fork 1 up twice**

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
10	31.84	3.02	0.03	0.01
15	49.30	3.34	0.05	0.02
20	69.99	3.70	0.06	0.03
25	95.14	3.96	0.07	0.07
30	122.84	4.38	0.09	0.06
35	153.69	4.59	0.12	0.08
40	188.82	5.04	0.13	0.12
45	227.54	5.36	0.15	0.16
50	268.92	5.52	0.19	0.17
55	316.25	5.83	0.21	0.20
60	365.58	6.10	0.24	0.24
65	420.03	6.35	0.28	0.29
70	475.78	6.89	0.31	0.33
75	536.58	7.12	0.33	0.39
80	599.24	7.38	0.37	0.43
85	665.07	7.68	0.42	0.48
90	737.27	8.02	0.46	0.53
95	808.87	8.36	0.48	0.62
100	891.40	8.58	0.53	0.68
105	973.23	8.80	0.59	0.73
110	1,050.41	9.36	0.61	0.82
115	1,151.94	9.78	0.65	0.91
120	1,247.31	9.79	0.73	0.94
125	1,343.68	10.22	0.76	1.04
130	1,440.24	10.56	0.85	1.16
135	1,549.18	10.71	0.93	1.21
140	1,658.09	11.10	0.94	1.30
145	1,781.20	11.42	1.01	1.39
150	1,915.78	11.83	1.09	1.49
155	1,997.38	11.92	1.12	1.64
160	2,128.31	12.32	1.17	1.72
165	2,257.28	12.76	1.25	1.83
170	2,375.21	13.35	1.32	1.92
175	2,506.76	13.10	1.43	1.95
180	2,656.16	13.92	1.48	2.13
185	2,814.12	13.74	1.56	2.24
190	2,952.26	13.90	1.63	2.49
195	3,099.22	14.54	1.69	2.53
200	3,253.16	15.37	1.74	2.67

Table 63: Time Information for **no fork 1 up twice**

C.3.3 Dictionary

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
2	11.18	2.32	0.02	0.02
3	14.51	2.44	0.02	0.04

Table 64: Time Information for **mutual exclusion**

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
4	18.04	2.52	0.03	0.10
5	21.61	2.66	0.04	0.19
6	25.69	2.75	0.05	0.33
7	29.59	3.01	0.06	0.48
8	34.05	3.05	0.07	0.70
9	38.21	3.13	0.08	0.95
10	42.81	3.24	0.09	1.23
15	68.70	3.69	0.19	3.16
20	98.67	4.19	0.34	5.97
25	133.01	4.55	0.60	9.70
30	173.66	5.09	0.85	14.46
35	217.20	5.54	1.19	20.40
40	264.11	6.03	1.65	26.43
45	318.70	6.55	2.09	33.78
50	375.78	6.91	3.10	41.97
55	438.45	7.35	3.99	51.45
60	504.92	7.71	4.63	61.43
65	577.75	8.15	5.68	72.54
70	654.31	8.56	6.56	83.96
75	745.31	9.51	7.91	96.91
80	822.56	9.66	9.27	110.98
85	909.93	10.04	10.80	125.23
90	1,009.68	10.28	12.83	142.14
95	1,103.25	11.16	18.04	158.32
100	1,223.70	11.58	20.29	175.99
105	1,326.20	12.18	23.76	195.03
110	1,433.38	12.58	26.07	212.95
115	1,557.64	12.95	29.75	232.66
120	1,674.93	13.54	32.90	253.95
125	1,804.49	14.17	37.17	275.26
130	1,935.10	14.55	40.70	296.87
135	2,087.34	15.16	45.61	322.47
140	2,223.43	15.90	49.95	347.28
145	2,377.10	16.66	54.19	370.99
150	2,587.45	17.94	61.62	405.03
155	2,671.02	17.67	63.94	428.19
160	2,932.09	18.05	70.36	455.47
165	3,025.15	18.74	77.32	484.93
170	3,196.60	19.53	82.73	514.57
175	3,363.80	19.87	89.46	542.99
180	3,569.91	20.36	125.07	582.80
185	3,759.29	20.90	104.73	610.39
190	3,938.50	21.83	142.67	642.98
195	4,141.20	22.14	151.44	675.77
200	4,363.02	22.41	160.24	721.54

Table 64: Time Information for **mutual exclusion**

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
2	11.18	2.30	0.01	0.02
3	14.51	2.40	0.02	0.02
4	18.04	2.45	0.02	0.02
5	21.61	2.61	0.04	0.02
6	25.69	2.71	0.04	0.03
7	29.59	2.83	0.05	0.05
8	34.05	2.97	0.07	0.05
9	38.21	3.15	0.08	0.07
10	42.81	3.23	0.09	0.08
15	68.70	3.68	0.19	0.20
20	98.67	4.08	0.33	0.35
25	133.01	4.48	0.58	0.55
30	173.66	5.06	0.84	0.81
35	217.20	5.48	1.18	1.09
40	264.11	5.96	1.62	1.48
45	318.70	6.45	2.07	1.86
50	375.78	7.03	3.11	2.34
55	438.45	7.43	3.74	2.78
60	504.92	7.65	4.68	3.29
65	577.75	8.09	5.63	3.85
70	654.31	8.40	6.49	4.79
75	745.31	9.13	8.13	4.98
80	822.56	9.61	9.21	6.28
85	909.93	10.06	10.70	6.43
90	1,009.68	10.38	12.16	7.63
95	1,103.25	10.74	18.74	7.86
100	1,223.70	11.56	20.83	9.58
105	1,326.20	12.06	23.66	10.04
110	1,433.38	12.45	26.92	11.07
115	1,557.64	12.97	29.77	12.40
120	1,674.93	13.51	33.29	13.67
125	1,804.49	14.04	36.13	14.92
130	1,935.10	14.50	40.86	16.06
135	2,087.34	15.10	45.21	16.15
140	2,223.43	16.01	49.59	18.67
145	2,377.10	16.59	54.51	19.72
150	2,587.45	17.47	58.92	21.84
155	2,671.02	17.55	64.14	22.76
160	2,932.09	18.06	70.93	23.48
165	3,025.15	18.78	76.42	24.80
170	3,196.60	18.98	82.79	26.82
175	3,363.80	19.78	90.50	27.10
180	3,569.91	20.48	96.47	29.89
185	3,759.29	20.75	104.65	33.30
190	3,938.50	22.22	143.26	33.39
195	4,141.20	22.47	148.99	35.83
200	4,363.02	22.72	160.40	38.57

Table 65: Time Information for **no fork 1 up twice**

C.3.4 Host

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
2	12.87	2.38	0.02	0.02
3	16.33	2.47	0.03	0.05
4	19.78	2.56	0.03	0.14
5	23.38	2.68	0.04	0.26
6	27.17	2.85	0.05	0.44
7	31.15	2.94	0.06	0.66
8	35.43	3.08	0.08	0.90
9	39.59	3.19	0.08	1.20
10	44.05	3.27	0.09	1.56
15	68.66	3.72	0.16	3.98
20	96.50	4.23	0.26	7.43
25	129.74	4.79	0.38	12.08
30	165.75	5.09	0.51	17.84
35	205.61	5.45	0.65	24.89
40	251.27	5.98	0.78	32.66
45	299.64	6.46	1.06	42.14
50	351.91	6.77	1.32	51.92
55	406.00	7.11	1.50	63.78
60	469.66	7.53	1.74	75.90
65	535.25	7.95	1.98	89.05
70	603.15	8.50	2.31	103.94
75	680.11	9.18	2.55	119.46
80	750.68	9.55	2.86	136.01
85	838.52	9.86	3.14	160.06
90	921.47	10.17	4.02	174.33
95	1,011.61	10.54	4.52	194.27
100	1,102.68	11.07	4.79	214.71
105	1,210.76	11.60	5.28	237.48
110	1,306.13	11.84	5.66	264.28
115	1,411.81	12.36	6.08	285.09
120	1,534.02	12.72	6.72	310.75
125	1,639.29	13.34	7.24	338.76
130	1,755.88	13.57	7.83	366.74
135	1,892.40	14.72	8.39	394.65
140	2,010.41	14.98	9.00	431.66
145	2,145.24	15.79	9.71	462.64
150	2332.00	17.11	10.93	498.65
155	2,427.08	16.70	10.68	527.11
160	2,566.14	17.21	11.54	557.10
165	2,716.69	17.60	12.14	592.18
170	2,883.65	18.12	12.87	629.50
175	3,026.86	18.68	15.65	669.98
180	3,194.43	19.34	16.67	704.91
185	3,359.89	19.96	17.32	747.95
190	3,545.29	20.35	18.43	791.54

Table 66: Time Information for **mutual exclusion**

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
195	3,709.43	20.82	18.95	829.79
200	3,896.34	21.64	20.07	874.60

Table 66: Time Information for **mutual exclusion**

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
2	12.87	2.40	0.02	0.02
3	16.33	2.40	0.02	0.02
4	19.78	2.53	0.03	0.01
5	23.38	2.64	0.03	0.04
6	27.17	2.77	0.05	0.04
7	31.15	2.93	0.06	0.05
8	35.43	3.01	0.07	0.07
9	39.59	3.16	0.07	0.10
10	44.05	3.28	0.09	0.11
15	68.66	3.61	0.16	0.26
20	96.50	4.16	0.24	0.48
25	129.74	4.46	0.37	0.77
30	165.75	5.01	0.49	1.07
35	205.61	5.40	0.63	1.46
40	251.27	5.92	0.78	1.96
45	299.64	6.35	1.04	2.49
50	351.91	6.78	1.25	3.10
55	406.00	7.22	1.48	3.80
60	469.66	7.43	1.68	4.45
65	535.25	8.09	1.89	5.33
70	603.15	8.45	2.15	6.11
75	680.11	8.83	2.51	6.99
80	750.68	9.31	2.78	7.94
85	838.52	9.64	3.11	9.24
90	921.47	10.10	3.86	10.13
95	1,011.61	10.47	4.30	11.29
100	1,102.68	10.77	4.76	12.56
105	1,210.76	11.18	5.20	13.72
110	1,306.13	11.74	5.65	15.14
115	1,411.81	12.20	6.13	16.56
120	1,534.02	12.58	6.60	18.07
125	1,639.29	13.15	7.15	19.56
130	1,755.88	13.76	7.67	21.19
135	1,892.40	14.63	8.25	22.77
140	2,010.41	15.14	8.80	24.68
145	2,145.24	15.59	9.41	26.46
150	2332.00	16.26	10.14	28.48
155	2,427.08	16.61	10.67	30.33
160	2,566.14	17.11	11.34	32.87
165	2,716.69	17.53	11.95	34.42
170	2,883.65	17.95	12.64	36.73

Table 67: Time Information for **no fork 1 up twice**

Philosophers	Language Processing	Artifact Building	MHP	State Propagation
175	3,026.86	18.90	15.37	38.69
180	3,194.43	19.39	16.16	41.16
185	3,359.89	19.47	17.15	43.05
190	3,545.29	20.87	17.62	45.44
195	3,709.43	21.11	18.83	47.94
200	3,896.34	21.06	19.57	50.06

Table 67: Time Information for **no fork 1 up twice**

C.4 Chiron

Artists	Language Processing	Artifact Building	MHP	State Propagation
2	25.92	2.85	0.04	0.02
3	33.00	3.10	0.05	0.04
4	41.11	3.43	0.08	0.06
5	50.31	3.80	0.11	0.08
6	59.45	3.97	0.13	0.11
7	69.91	4.32	0.17	0.15
8	81.35	4.52	0.22	0.18
9	95.16	4.70	0.27	0.21
10	106.35	5.02	0.33	0.28
11	119.87	5.30	0.40	0.32
12	134.56	5.54	0.45	0.37
13	150.03	5.94	0.56	0.44
14	166.55	6.03	0.65	0.50
15	183.17	6.51	0.71	0.59
16	200.19	6.97	0.88	0.65
17	222.63	6.91	0.98	0.75
18	248.79	7.25	1.11	0.81
19	265.57	7.55	1.27	0.93
20	283.76	7.83	1.44	1.00
21	304.49	8.09	1.59	1.08
22	327.57	8.38	1.81	1.23
23	352.27	8.88	1.99	1.37
24	376.60	9.17	2.18	1.51
25	401.20	9.27	2.47	1.72
26	438.95	9.73	2.71	1.68
27	471.86	10.12	2.94	1.81
28	500.47	10.38	3.31	1.94
29	523.46	10.62	3.54	2.12
30	548.40	10.89	3.76	2.26
31	575.73	11.12	4.22	2.45
32	605.15	11.55	4.68	2.47
33	639.57	11.70	4.91	2.84
34	679.16	12.14	5.47	2.85
35	735.56	12.47	5.73	3.09
36	777.48	12.80	6.14	3.29

Table 68: Time Information for **p01**

Artists	Language Processing	Artifact Building	MHP	State Propagation
37	816.58	13.12	6.87	3.48
38	844.15	13.26	7.21	3.81
39	877.52	13.74	7.68	3.91
40	903.15	13.89	8.41	4.03
41	937.24	14.29	9.05	4.02
42	976.63	14.70	9.62	4.29
43	1,039.52	14.79	10.18	4.71
44	1,103.74	15.25	11.13	4.49
45	1,155.19	15.48	11.37	5.47
46	1,203.26	16.43	12.53	5.50
47	1,246.21	16.92	13.36	5.60
48	1,283.97	17.09	14.22	5.88
49	1,319.62	18.39	15.10	6.13
50	1,357.23	18.61	15.61	6.14
51	1,410.04	18.97	17.15	6.96
52	1,497.88	19.20	17.90	7.09
53	1,559.43	19.56	18.60	7.92

Table 68: Time Information for **p01**

Artists	Language Processing	Artifact Building	MHP	State Propagation
2	25.92	4.02	0.06	0.22
3	33.00	4.54	0.08	0.39
4	41.11	5.22	0.11	0.63
5	50.31	5.63	0.14	0.87
6	59.45	6.41	0.17	1.16
7	69.91	6.82	0.23	1.53
8	81.35	7.25	0.27	1.92
9	95.16	7.86	0.32	2.32
10	106.35	8.62	0.39	2.89
11	119.87	9.39	0.47	3.33
12	134.56	10.10	0.56	3.85
13	150.03	10.82	0.65	4.57
14	166.55	11.63	0.74	5.18
15	183.17	12.13	0.84	5.80
16	200.19	13.20	1.01	6.72
17	222.63	13.86	1.17	7.38
18	248.79	14.55	1.26	8.17
19	265.57	15.36	1.46	9.13
20	283.76	16.33	1.60	10.03
21	304.49	16.94	1.78	11.00
22	327.57	18.52	2.03	12.04
23	352.27	18.94	2.24	13.05
24	376.60	19.93	2.43	14.25
25	401.20	21.22	2.76	15.33
26	438.95	21.99	3.07	16.54
27	471.86	23.35	3.23	17.68

Table 69: Time Information for **p02**

Artists	Language Processing	Artifact Building	MHP	State Propagation
28	500.47	24.55	3.63	19.00
29	523.46	25.75	3.94	20.41
30	548.40	26.45	4.17	21.71
31	575.73	27.89	4.74	23.69
32	605.15	28.88	5.14	24.39
33	639.57	29.75	5.22	26.09
34	679.16	30.69	6.03	27.58
35	735.56	35.13	6.41	29.26
36	777.48	33.83	6.78	30.62
37	816.58	35.51	7.42	32.45
38	844.15	36.30	7.93	33.93
39	877.52	37.89	8.23	36.03
40	903.15	39.35	9.05	38.32
41	937.24	40.70	9.65	40.75
42	976.63	42.22	10.23	41.32
43	1,039.52	43.70	11.16	43.48
44	1,103.74	46.90	11.77	46.11
45	1,155.19	46.65	12.29	47.52
46	1,203.26	48.95	13.60	49.53
47	1,246.21	49.28	14.31	51.66
48	1,283.97	52.08	15.18	53.89
49	1,319.62	53.50	16.10	56.35
50	1,357.23	55.15	17.24	58.55
51	1,410.04	56.64	18.25	60.55
52	1,497.88	58.63	19.15	72.31
53	1,559.43	60.51	20.28	65.80

Table 69: Time Information for **p02**

Artists	Language Processing	Artifact Building	MHP	State Propagation
2	25.92	2.98	0.04	0.03
3	33.00	3.27	0.06	0.04
4	41.11	3.52	0.09	0.07
5	50.31	3.89	0.11	0.10
6	59.45	4.05	0.14	0.14
7	69.91	4.39	0.18	0.18
8	81.35	4.60	0.22	0.23
9	95.16	4.79	0.28	0.27
10	106.35	5.12	0.34	0.34
11	119.87	5.44	0.39	0.42
12	134.56	5.61	0.46	0.46
13	150.03	5.86	0.56	0.53
14	166.55	6.05	0.63	0.65
15	183.17	6.78	0.73	0.71
16	200.19	7.05	0.87	0.81
17	222.63	7.20	0.99	0.89
18	248.79	7.52	1.07	1.04

Table 70: Time Information for **p03**

Artists	Language Processing	Artifact Building	MHP	State Propagation
19	265.57	7.79	1.29	1.09
20	283.76	8.06	1.42	1.21
21	304.49	8.27	1.55	1.36
22	327.57	8.63	1.84	1.47
23	352.27	9.19	1.98	1.62
24	376.60	9.58	2.18	1.73
25	401.20	9.65	2.49	1.81
26	438.95	9.90	2.66	2.11
27	471.86	10.40	2.90	2.22
28	500.47	10.78	3.30	2.31
29	523.46	10.95	3.50	2.54
30	548.40	11.15	3.84	2.65
31	575.73	11.70	4.27	2.93
32	605.15	11.80	4.67	3.09
33	639.57	11.98	5.01	3.25
34	679.16	12.40	5.47	3.29
35	735.56	12.62	5.79	3.69
36	777.48	12.80	6.24	3.88
37	816.58	13.18	6.58	4.11
38	844.15	13.46	7.23	4.36
39	877.52	13.87	7.57	4.69
40	903.15	14.32	8.47	4.54
41	937.24	14.57	8.90	5.20
42	976.63	14.81	9.36	5.47
43	1,039.52	15.09	10.21	5.63
44	1,103.74	15.51	11.13	5.45
45	1,155.19	16.02	11.41	6.27
46	1,203.26	16.98	12.61	6.34
47	1,246.21	17.39	13.37	6.59
48	1,283.97	17.68	14.17	6.62
49	1,319.62	18.73	15.14	6.97
50	1,357.23	19.08	15.87	7.36
51	1,410.04	19.11	17.03	7.58
52	1,497.88	19.46	17.80	8.34
53	1,559.43	20.69	18.80	8.66

Table 70: Time Information for **p03**

Artists	Language Processing	Artifact Building	MHP	State Propagation
2	25.92	2.93	0.04	0.02
3	33.00	3.26	0.06	0.03
4	41.11	3.55	0.08	0.07
5	50.31	3.76	0.11	0.10
6	59.45	4.17	0.13	0.14
7	69.91	4.42	0.18	0.19
8	81.35	4.66	0.22	0.24
9	95.16	4.90	0.27	0.29

Table 71: Time Information for **p04**

Artists	Language Processing	Artifact Building	MHP	State Propagation
10	106.35	5.18	0.34	0.38
11	119.87	5.45	0.40	0.43
12	134.56	5.81	0.47	0.50
13	150.03	6.16	0.55	0.64
14	166.55	6.32	0.65	0.67
15	183.17	6.61	0.74	0.77
16	200.19	6.95	0.90	0.90
17	222.63	7.24	1.00	1.00
18	248.79	7.49	1.12	1.13
19	265.57	7.67	1.33	1.25
20	283.76	8.44	1.47	1.38
21	304.49	8.48	1.60	1.50
22	327.57	9.03	1.87	1.68
23	352.27	9.40	2.04	1.84
24	376.60	9.57	2.24	1.94
25	401.20	9.84	2.51	2.24
26	438.95	10.08	2.78	2.42
27	471.86	10.22	3.00	2.48
28	500.47	10.51	3.36	2.82
29	523.46	10.78	3.68	3.05
30	548.40	11.24	3.86	3.32
31	575.73	11.36	4.30	3.61
32	605.15	11.99	4.81	3.65
33	639.57	12.44	4.98	4.00
34	679.16	12.44	5.51	4.36
35	735.56	12.30	5.88	4.53
36	777.48	13.04	6.33	4.89
37	816.58	13.15	7.01	5.08
38	844.15	13.43	7.49	5.30
39	877.52	14.06	7.92	5.71
40	903.15	14.71	8.82	5.80
41	937.24	14.98	9.25	6.34
42	976.63	15.49	9.75	6.60
43	1,039.52	16.07	11.10	6.58
44	1,103.74	16.10	11.53	7.17
45	1,155.19	16.59	11.93	7.76
46	1,203.26	17.46	12.97	8.19
47	1,246.21	17.58	13.87	8.59
48	1,283.97	17.87	14.40	8.76
49	1,319.62	18.43	15.62	9.50
50	1,357.23	18.96	16.40	9.81
51	1,410.04	19.20	16.91	10.52
52	1,497.88	19.68	18.43	10.78
53	1,559.43	20.33	19.38	11.58

Table 71: Time Information for **p04**

Artists	Language Processing	Artifact Building	MHP	State Propagation
2	25.92	3.89	0.04	0.11
3	33.00	4.17	0.06	0.25
4	41.11	4.56	0.09	0.56
5	50.31	4.68	0.11	0.91
6	59.45	5.09	0.14	1.39
7	69.91	5.36	0.17	2.22
8	81.35	5.61	0.22	3.10
9	95.16	5.93	0.26	4.14
10	106.35	6.20	0.34	5.72
11	119.87	6.64	0.40	7.41
12	134.56	6.84	0.45	9.14
13	150.03	7.13	0.56	11.73
14	166.55	7.48	0.65	14.33
15	183.17	7.84	0.72	17.13
16	200.19	8.29	0.88	21.12
17	222.63	8.21	0.98	24.80
18	248.79	8.56	1.08	29.11
19	265.57	9.10	1.28	34.28
20	283.76	9.43	1.46	39.59
21	304.49	9.59	1.55	45.17
22	327.57	10.06	1.82	52.06
23	352.27	10.25	1.99	58.92
24	376.60	10.68	2.18	65.88
25	401.20	10.84	2.47	75.35
26	438.95	11.23	2.69	84.48
27	471.86	11.50	2.88	93.01
28	500.47	12.22	3.31	104.54
29	523.46	12.39	3.51	115.58
30	548.40	12.54	3.78	126.39
31	575.73	12.99	4.13	141.51
32	605.15	13.06	4.67	154.95
33	639.57	13.28	4.98	168.56
34	679.16	13.62	5.36	185.74
35	735.56	13.97	5.76	201.54
36	777.48	13.33	6.14	218.18
37	816.58	13.63	6.66	238.13
38	844.15	13.99	7.26	256.90
39	877.52	14.10	7.81	275.80
40	903.15	14.65	8.51	299.67
41	937.24	15.50	8.91	321.03
42	976.63	15.95	9.28	342.03
43	1,039.52	16.21	10.16	373.43
44	1,103.74	16.47	11.16	394.99
45	1,155.19	17.10	11.78	419.03
46	1,203.26	17.26	12.53	450.79
47	1,246.21	17.68	13.43	478.39
48	1,283.97	17.90	14.27	504.74

Table 72: Time Information for **p05**

Artists	Language Processing	Artifact Building	MHP	State Propagation
49	1,319.62	18.52	15.10	540.93
50	1,357.23	18.79	15.76	572.99
51	1,410.04	19.22	17.18	605.57
52	1,497.88	19.47	17.71	647.38
53	1,559.43	19.79	19.02	678.12

Table 72: Time Information for **p05**

Artists	Language Processing	Artifact Building	MHP	State Propagation
2	25.92	2.97	0.04	0.02
3	33.00	3.21	0.05	0.04
4	41.11	3.59	0.08	0.07
5	50.31	3.73	0.11	0.08
6	59.45	4.09	0.14	0.10
7	69.91	4.39	0.18	0.15
8	81.35	4.61	0.22	0.19
9	95.16	4.88	0.27	0.21
10	106.35	5.23	0.33	0.27
11	119.87	5.61	0.40	0.32
12	134.56	5.88	0.45	0.38
13	150.03	6.23	0.56	0.44
14	166.55	6.46	0.64	0.52
15	183.17	6.75	0.72	0.58
16	200.19	7.06	0.87	0.66
17	222.63	7.34	0.99	0.74
18	248.79	7.61	1.10	0.79
19	265.57	8.10	1.27	0.93
20	283.76	8.67	1.40	1.04
21	304.49	8.73	1.56	1.11
22	327.57	9.09	1.79	1.23
23	352.27	9.34	1.97	1.33
24	376.60	10.03	2.16	1.47
25	401.20	9.86	2.46	1.53
26	438.95	10.19	2.69	1.69
27	471.86	10.74	2.92	1.79
28	500.47	10.61	3.29	1.97
29	523.46	11.26	3.54	2.20
30	548.40	11.64	3.80	2.18
31	575.73	11.90	4.24	2.29
32	605.15	12.21	4.56	2.55
33	639.57	12.40	4.88	2.76
34	679.16	13.10	5.41	2.90
35	735.56	13.55	5.64	3.19
36	777.48	13.66	6.13	3.39
37	816.58	14.18	6.71	3.59
38	844.15	14.37	7.19	3.59
39	877.52	14.48	7.62	3.93

Table 73: Time Information for **p06**

Artists	Language Processing	Artifact Building	MHP	State Propagation
40	903.15	14.59	8.39	4.00
41	937.24	14.82	8.83	4.26
42	976.63	15.07	9.70	4.02
43	1,039.52	15.47	10.34	4.30
44	1,103.74	15.95	11.07	4.85
45	1,155.19	16.47	11.65	4.75
46	1,203.26	17.17	12.77	5.14
47	1,246.21	17.77	13.30	5.43
48	1,283.97	17.61	14.12	5.36
49	1,319.62	18.42	15.18	6.07
50	1,357.23	19.32	15.99	6.51
51	1,410.04	19.67	16.81	6.27
52	1,497.88	20.47	17.47	7.16
53	1,559.43	19.85	19.37	7.06

Table 73: Time Information for **p06**

Artists	Language Processing	Artifact Building	MHP	State Propagation
2	25.92	4.48	0.06	0.13
3	33.00	4.77	0.08	0.21
4	41.11	5.20	0.11	0.35
5	50.31	5.48	0.14	0.46
6	59.45	5.77	0.16	0.63
7	69.91	6.14	0.21	0.84
8	81.35	6.41	0.26	1.01
9	95.16	6.73	0.31	1.25
10	106.35	6.94	0.40	1.52
11	119.87	7.28	0.47	1.75
12	134.56	7.46	0.54	2.02
13	150.03	7.98	0.64	2.42
14	166.55	8.21	0.74	2.74
15	183.17	8.53	0.83	3.06
16	200.19	8.88	0.97	3.56
17	222.63	9.17	1.13	3.89
18	248.79	9.51	1.25	4.27
19	265.57	9.69	1.43	4.82
20	283.76	9.98	1.62	5.31
21	304.49	10.31	1.75	5.73
22	327.57	10.93	2.02	6.41
23	352.27	11.17	2.44	6.94
24	376.60	11.30	2.41	7.36
25	401.20	11.66	2.72	8.07
26	438.95	12.09	2.97	8.73
27	471.86	12.83	3.19	9.33
28	500.47	12.96	3.62	9.90
29	523.46	13.20	3.97	10.68
30	548.40	14.01	4.11	11.40

Table 74: Time Information for **p07**

Artists	Language Processing	Artifact Building	MHP	State Propagation
31	575.73	13.75	4.61	12.20
32	605.15	14.23	4.93	13.06
33	639.57	14.55	5.24	13.72
34	679.16	14.77	5.84	14.57
35	735.56	15.08	6.29	15.48
36	777.48	15.65	6.63	16.24
37	816.58	15.80	7.31	17.06
38	844.15	16.04	7.95	18.64
39	877.52	16.34	8.42	18.71
40	903.15	16.72	8.94	19.88
41	937.24	16.87	9.65	21.01
42	976.63	17.64	10.05	21.95
43	1,039.52	17.59	11.26	22.81
44	1,103.74	18.33	11.87	23.72
45	1,155.19	18.80	12.49	25.07
46	1,203.26	18.62	13.45	26.14
47	1,246.21	19.90	14.21	27.39
48	1,283.97	20.04	15.25	28.84
49	1,319.62	20.42	16.35	29.57
50	1,357.23	20.96	17.21	31.02
51	1,410.04	21.19	17.49	32.42
52	1,497.88	22.27	18.70	33.91
53	1,559.43	21.92	19.89	34.84

Table 74: Time Information for **p07**

Artists	Language Processing	Artifact Building	MHP	State Propagation
2	25.92	3.42	0.06	0.23
3	33.00	3.69	0.08	0.38
4	41.11	4.20	0.11	0.64
5	50.31	4.34	0.15	0.86
6	59.45	4.62	0.18	1.14
7	69.91	4.93	0.22	1.55
8	81.35	5.17	0.28	1.91
9	95.16	5.54	0.32	2.34
10	106.35	5.83	0.41	2.86
11	119.87	6.21	0.47	3.36
12	134.56	6.43	0.55	3.87
13	150.03	6.76	0.67	4.54
14	166.55	7.10	0.77	5.14
15	183.17	7.33	0.84	5.87
16	200.19	7.75	1.01	6.67
17	222.63	7.96	1.13	7.38
18	248.79	8.19	1.27	8.11
19	265.57	8.39	1.47	9.25
20	283.76	8.78	1.62	10.00
21	304.49	9.33	1.78	11.44

Table 75: Time Information for **p08**

Artists	Language Processing	Artifact Building	MHP	State Propagation
22	327.57	9.77	2.05	12.19
23	352.27	9.86	2.25	13.06
24	376.60	10.41	2.45	14.17
25	401.20	10.50	2.79	15.36
26	438.95	11.05	3.05	16.40
27	471.86	11.35	3.29	17.54
28	500.47	11.60	3.62	19.13
29	523.46	12.37	4.02	20.34
30	548.40	12.36	4.24	21.55
31	575.73	12.66	4.58	23.32
32	605.15	12.92	5.01	24.62
33	639.57	13.17	5.54	25.75
34	679.16	13.49	5.99	27.64
35	735.56	13.70	6.38	29.11
36	777.48	14.05	6.65	30.63
37	816.58	15.21	7.27	32.87
38	844.15	15.31	8.01	34.02
39	877.52	15.57	8.47	35.73
40	903.15	15.93	9.18	38.47
41	937.24	16.35	9.72	40.01
42	976.63	16.92	10.20	41.05
43	1,039.52	17.12	11.09	43.35
44	1,103.74	17.43	11.75	45.73
45	1,155.19	17.91	12.58	47.62
46	1,203.26	18.98	13.61	49.81
47	1,246.21	19.05	14.43	51.57
48	1,283.97	19.72	15.30	53.94
49	1,319.62	19.93	16.35	56.35
50	1,357.23	19.97	16.66	58.18
51	1,410.04	21.17	18.02	60.34
52	1,497.88	21.20	19.06	63.36
53	1,559.43	21.33	20.00	66.32

Table 75: Time Information for **p08**

Artists	Language Processing	Artifact Building	MHP	State Propagation
2	25.92	2.94	0.04	0.17
3	33.00	3.43	0.06	0.40
4	41.11	3.93	0.08	0.97
5	50.31	4.53	0.11	1.67
6	59.45	5.16	0.14	2.63
7	69.91	6.19	0.18	4.47
8	81.35	7.60	0.22	6.45
9	95.16	9.29	0.26	8.79
10	106.35	10.98	0.33	12.94
11	119.87	13.82	0.40	17.05
12	134.56	17.04	0.46	21.57

Table 76: Time Information for **p09**

Artists	Language Processing	Artifact Building	MHP	State Propagation
13	150.03	22.16	0.56	29.46
14	166.55	27.82	0.66	36.57
15	183.17	33.62	0.73	44.36
16	200.19	41.13	0.87	57.45
17	222.63	50.41	0.99	68.88
18	248.79	62.27	1.11	81.58
19	265.57	75.11	1.28	101.33
20	283.76	87.47	1.45	118.36
21	304.49	108.19	1.61	136.69
22	327.57	126.75	1.80	166.55
23	352.27	146.50	2.00	191.00
24	376.60	173.60	2.20	216.71
25	401.20	202.15	2.50	257.35
26	438.95	226.92	2.74	290.15
27	471.86	262.87	2.97	325.25
28	500.47	312.53	3.34	381.53
29	523.46	351.55	3.64	427.48
30	548.40	403.72	3.86	474.77
31	575.73	463.19	4.21	544.74
32	605.15	518.15	4.67	601.98
33	639.57	588.80	4.86	662.02
34	679.16	661.63	5.49	753.76
35	735.56	732.70	6.07	830.42
36	777.48	823.63	6.27	906.88
37	816.58	912.85	6.74	1,034.28
38	844.15	1,013.20	7.18	1,115.53
39	877.52	1,118.63	7.81	1,206.09
40	903.15	1,223.29	8.73	1,348.02
41	937.24	1,343.82	8.94	1,472.28
42	976.63	1,479.51	9.60	1,584.34
43	1,039.52	1,605.99	10.40	1,755.51
44	1,103.74	1,771.38	11.30	1,886.70
45	1,155.19	1,885.73	11.78	2,027.31
46	1,203.26	2,049.64	12.60	2,237.45
47	1,246.21	2,211.06	13.61	2,407.69
48	1,283.97	2,340.59	14.23	2,574.30
49	1,319.62	2,503.58	15.13	2,816.58
50	1,357.23	2,718.73	16.34	3,041.40
51	1,410.04	2,900.59	17.14	3,206.87
52	1,497.88	3,150.98	18.71	3,507.05
53	1,559.43	3,359.25	19.27	3,748.24

Table 76: Time Information for **p09**